ИЗУЧАЕМ REACT

Практическое руководство по созданию веб-приложений при помощи React и Redux

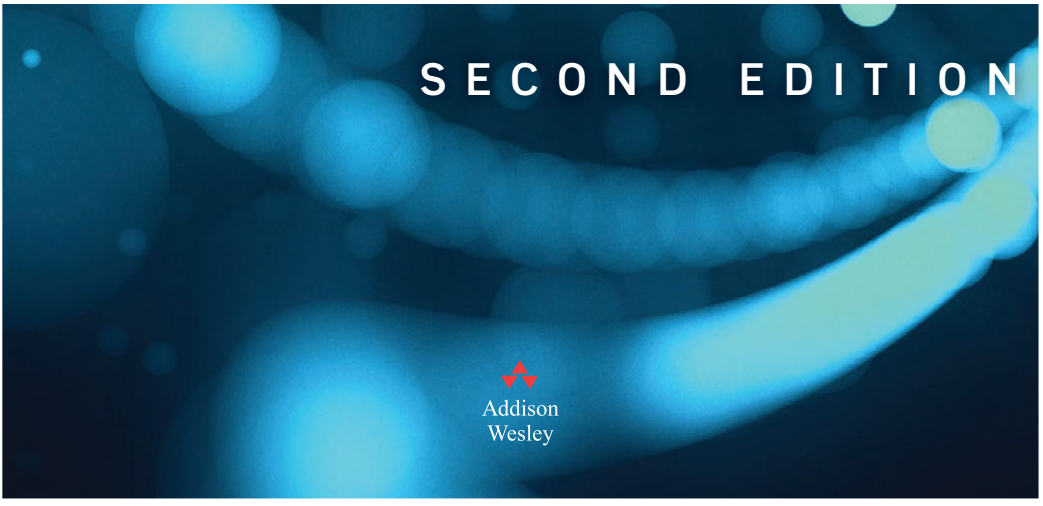
ВТОРОЕ ИЗДАНИЕ

КИРУПА ЧИННАТАМБИ


Мировой
компьютерный
бестселлер

LEARNING REACT

A Hands-On Guide to Building Web Applications Using React and Redux



ИЗУЧАЕМ REACT

Практическое руководство по созданию веб-приложений при помощи React и Redux

Москва
2019

УДК 004.4
ББК 32.973.2-018.2
Ч-63

Kirupa Chinnathambi
LEARNING REACT:
A Hands-on Guide to Building web applications using react and redux, 2nd ed

Чиннатамби, Кирупа.

Ч-63 Изучаем React / Кирупа Чиннатамби ; [пер. с англ. М. А. Райтмана]. — 2-е изд. — Москва : Эксмо, 2019. — 368 с. : ил. — (Мировой компьютерный бестселлер).

ISBN 978-5-04-098028-4

Второе, обновленное, издание снискавшей множество положительных отзывов на Amazon книги известного преподавателя основ веб-разработки, автора обучающего Youtube-канала Кирупы Чиннатамби. Эта книга позволит вам освоить разработку современных веб-приложений с использованием React и Redux. До выхода этой книги считалось, что освоить React самостоятельно, да еще и новичку, практически невозможно. Однако благодаря свежему взгляду и легкому стилю изложения автора этой книги сотни веб-разработчиков по всему миру признали, что освоили тему легко и быстро. На то, чтобы приступить к созданию первых собственных приложений на React, вам потребуется буквально несколько минут чтения. Дерзайте!

УДК 004.4
ББК 32.973.2-018.2

ISBN 978-5-04-098028-4

Посвящается моему отцу — он всегда верил в меня,
даже если то, что я делал, не имело для него
(а порой и меня самого) никакого смысла!

Содержание

Об авторе

Кирупа Чиннатамби потратил большую часть своей жизни на то, чтобы помочь другим людям полюбить веб-разработку так, как ее любит он.

В 1999 году, еще до появления слова «блог», он начал публиковать обучающие материалы на сайте **kirupa.com**. С тех пор он написал сотни статей, стал автором несколько книг (не таких хороших, как эта, разумеется!), а также создал множество видеороликов, которые вы можете найти на YouTube. Когда он не пишет и не говорит о веб-разработке, он тратит свои часы бодрствования на развитие Всемирной паутины, работая в качестве менеджера проектов в компании Microsoft. В часы небодрствования он, вероятно, спит или пишет о самом себе в третьем лице.

Вы можете найти его в Twitter (**twitter.com/kirupa**), Facebook (**facebook.com/kirupa**) или связаться с ним по электронной почте (**kirupa@kirupa.com**). Не стесняйтесь обращаться к нему в любое время.

Благодарности

Во-первых, эта книга не увидела бы свет без одобрения и поддержки моей замечательной жены, **Мины**. Если бы она не отсрочила достижение собственных целей, позволив мне потратить шесть месяцев на обдумывание, написание и доработку того, что вы здесь видите, то создание этой книги так и осталось бы мечтой.

Также я хотел бы поблагодарить **своих родителей.** Они всегда поощряли мои бесцельные поиски и позволяли делать то, что мне нравится, например, учить незнакомцев через Интернет делать классные вещи с помощью программирования, чем я занимался в конце 1990-х годов. Если бы не они, я не стал бы и вполовину таким суровым домоседом/ученым/воином, каким являюсь сегодня.

Написать слова, которые вы читаете, довольно легко, однако чрезвычайно сложно сделать так, чтобы эта книга попала к вам в руки. Чем больше я узнаю о деталях этого сложного механизма, тем больше восхищаюсь людьми, которые неустанно работают, следя за тем, чтобы он функционировал, как надо. Я благодарю **каждого сотрудника издательства Pearson**, работавшего над этим проектом! Тем не менее есть несколько человек, которых я хотел бы поблагодарить особо. Во-первых, я благодарю **Марка Тэйбера** за предоставленные возможности для совместной работы, **Криса Зана** за терпеливое решение моих многочисленных проблем, **Кристу Хэнсинг** за преобразование моей версии английского языка в нечто понятное, а также **Лоретту Йетс** за то, что когда-то она помогла наладить связи, позволившие воплотить этот проект. Техническая составляющая книги была тщательно проверена моими давними друзьями и сотрудниками **Кайлом Мюрреем (a. k. a. Krilnon)** и **Тревором МакКоли (a. k. a. senocular)**. Я бесконечно благодарен им за их подробные (и часто очень забавные!) комментарии.

Глава 1
Знакомство с React

Если на мгновение забыть об общем улучшении *внешнего вида* и *опыта использования* веб-приложения, то в этой сфере можно заметить фундаментальное изменение. Теперь веб-приложения проектируются и создаются иначе. Чтобы убедиться в этом, рассмотрим приложение, изображенное на рис. 1.1.

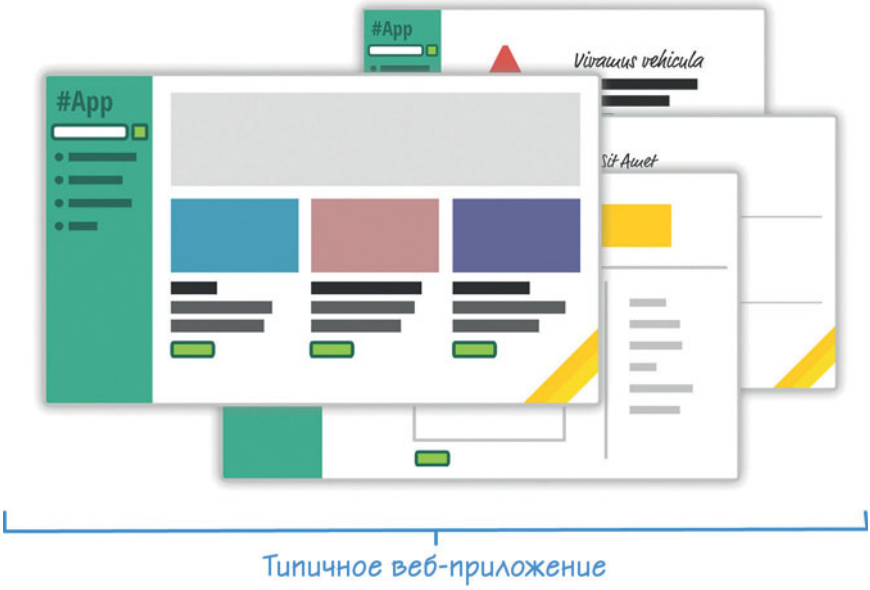


Рис. 1.1. Приложение

Данное приложение представляет собой каталог. Как и в случае с любой подобной программой, у нас есть обычный набор страниц, связанных с домашней страницей, страница результатов поиска, страница сведений и т. д. В следующих разделах мы рассмотрим два

подхода к созданию этого приложения. Да, по таинственному совпадению, мы в то же время познакомимся и с библиотекой React.

Вперед!

Многостраничный дизайн старой школы

Если бы вы создавали это приложение несколько лет назад, вероятно, применили бы подход, предполагающий использование нескольких отдельных страниц. В этом случае поток выглядел бы как на рис. 1.2.

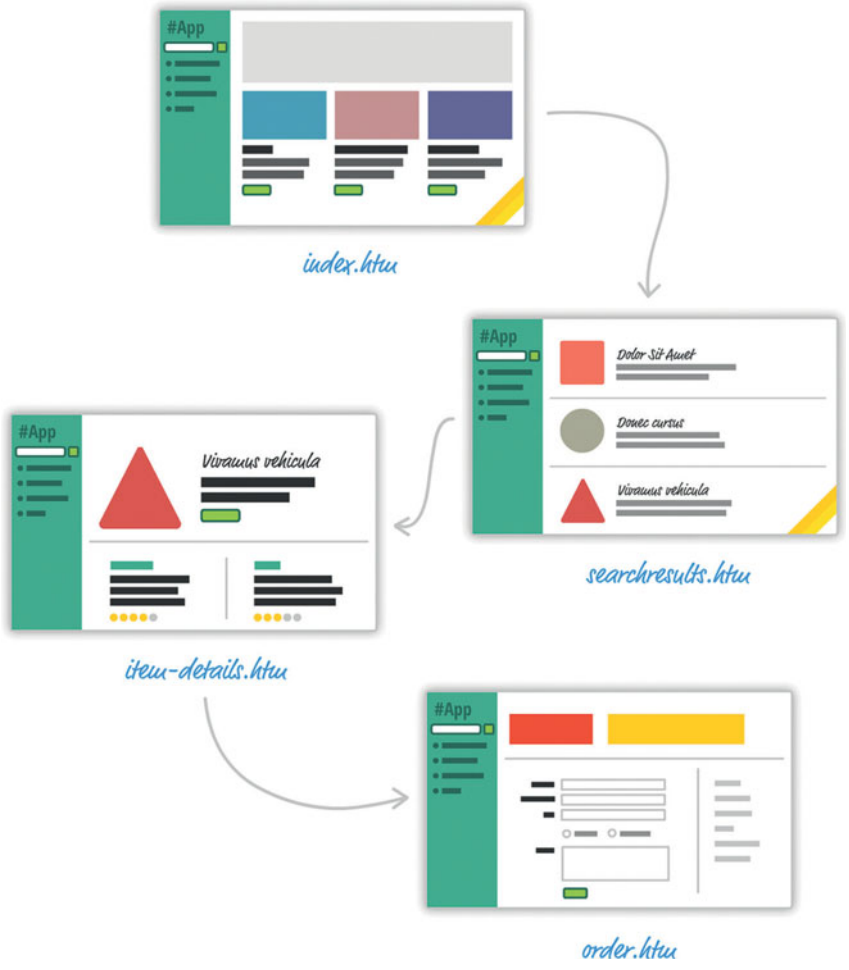


Рис. 1.2. Многостраничный дизайн

В ответ почти на каждое действие, изменяющее содержимое окна браузера, веб-приложение переводит вас на *совершенно другую страницу*. Помимо того, что уничтожение и воссоздание страниц ухудшает пользовательский опыт, это оказывает большое влияние на способ поддержания состояния приложения. За исключением сохранения пользовательских данных с помощью cookie-файлов и некоторых серверных механизмов вам не о чем беспокоиться. Жизнь хороша.

Одностраничные приложения новой школы

Сегодня модель веб-приложения, которая требует перехода между отдельными страницами, устаревает (см. рис. 1.3).

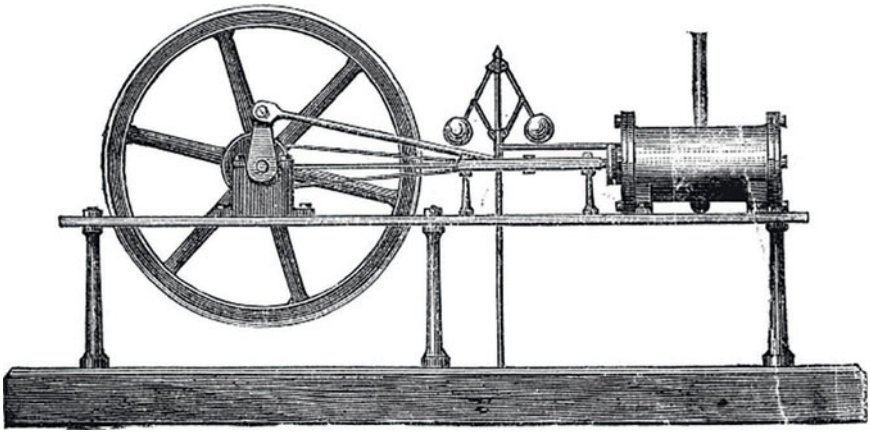


Рис. 1.3. Модель, использующая отдельные страницы, устарела, как этот паровой двигатель

Вместо этого современные приложения обычно основываются на так называемой **модели SPA** (Single-Page Applications, одностраничные приложения). Она позволяет создать приложение, в котором вы никогда не переходите на другие страницы и даже не перезагружаете их. При ее использовании различные представления приложения загружаются в одной и той же странице.

В случае с нашим приложением это выглядит как на рис. 1.4.

Рис. 1.4. Одностраничное приложение

В процессе взаимодействия пользователей с приложением мы заменяем содержимое области, обведенной красной пунктирной линией, данными и HTML-кодом, который соответствует тому, что пытается сделать пользователь. Это дает гораздо более плавный пользовательский опыт. Вы можете применять множество визуальных приемов для плавного перехода между элементами контента, как в тех классных приложениях, с которыми вы работаете на своем мобильном устройстве или настольном компьютере. Такие вещи невозможны, когда вы переходите со страницы на страницу.

Все это кажется непонятным, если вы никогда не слышали об одностраничных приложениях, но, вероятно, вы уже с ними сталкивались. Если вы использовали такие популярные веб-приложения, как Gmail, Facebook, Instagram или Twitter, то знайте: вы использовали

одностраничное приложение. Во всех этих приложениях контент отображается динамически, не требуя обновления или перехода на другую страницу.

Одностраничные приложения могут показаться сложными. Это *совсем* не так. Благодаря большому количеству улучшений как в языке JavaScript, так и в различных сторонних фреймворках и библиотеках, создание одностраничных приложений никогда не было таким простым. Но возможности для улучшения не закончились.

При создании одностраничных приложений вы однажды столкнетесь с тремя основными проблемами:

1. **В одностраничном приложении бóльшая часть времени уходит на синхронизацию данных с пользовательским интерфейсом.** Например, что вы сделаете, когда пользователь загрузит новое содержимое? Очистите поле поиска? Оставите активную вкладку по-прежнему видимой на навигационном элементе? Какие элементы вы сохраните на странице, а какие уничтожите?

   Эти вопросы актуальны именно для одностраничных приложений. При навигации между страницами в старой модели мы предполагали, что все элементы пользовательского интерфейса будут уничтожены и созданы заново. Это никогда не было проблемой.

2. **Манипулирование DOM* происходит очень медленно.** На запрашивание элементов вручную, добавление дочерних элементов (см. рис. 1.5), удаление поддеревьев и выполнение других операций с DOM в браузере тратится больше всего времени. К сожалению, в одностраничном приложении вам придется часто этим заниматься. Манипулирование DOM — это основной способ, позволяющий вам реагировать на действия пользователя и отображать новое содержимое.

---

* Document object model — объектная модель документа (*прим. перев.*).

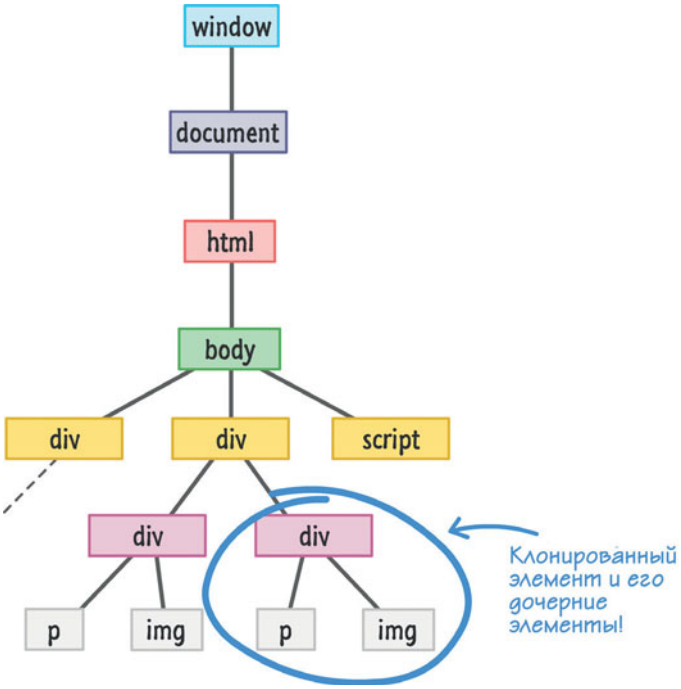


Рис. 1.5. Добавление дочерних элементов

3. **При работе с HTML-шаблонами могут возникать трудности.** В одностраничном приложении навигация представляет собой использование фрагментов HTML-кода для представления того, что необходимо отобразить. Эти фрагменты HTML-кода часто называются **шаблонами**, а процесс управления ими и наполнение данными с помощью JavaScript очень быстро усложняется.

   Еще хуже то, что внешний вид шаблонов и способ их взаимодействия с данными может сильно различаться в зависимости от используемого фреймворка. Например, вот как выглядит объявление и использование шаблона в шаблонизаторе Mustache:

```
var view = {
  title: "Макс",
  calc: function() {
    return 2 + 4;
  }
```

```
};

var output = Mustache.render("{{title}} заработал
{{calc}}", view);
```

Иногда шаблоны представляют собой чистый HTML-код, который можно с гордостью кому-нибудь продемонстрировать. В других случаях шаблоны могут быть непонятны, заполнены пользовательскими тегами, предназначенными для соотнесения HTML-элементов с некоторыми данными.

Несмотря на эти недостатки, одностраничные приложения никуда не денутся. Они являются частью настоящего и формируют будущее разработки веб-приложений. Это не значит, что перечисленные недостатки нужно терпеть. Читайте дальше.

Знакомьтесь с React

Facebook и Instagram решили, что всему есть предел. Опираясь на свой огромный опыт работы с одностраничными приложениями, они выпустили библиотеку под названием **React**, чтобы не только устранить описанные ранее недостатки, но и изменить сам подход к разработке одностраничных приложений.

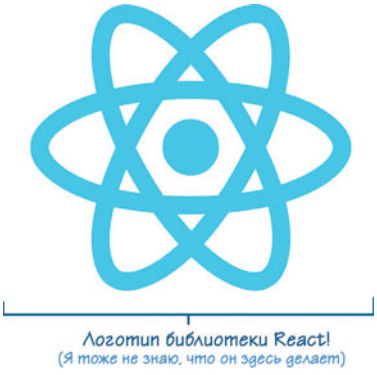


Рис. 1.6. Логотип библиотеки React!

В следующих разделах описаны преимущества этой библиотеки.

Автоматическое управление состоянием пользовательского интерфейса

При работе с одностраничными приложениями отслеживание и поддержание состояния пользовательского интерфейса требует больших усилий и времени. С библиотекой React вам придется беспокоиться только о конечном состоянии пользовательского интерфейса. Не важно, каким было его начальное состояние. Не имеет значения, какие шаги пользователей привели к его изменению. Важно лишь его конечное состояние (см. рис. 1.7).

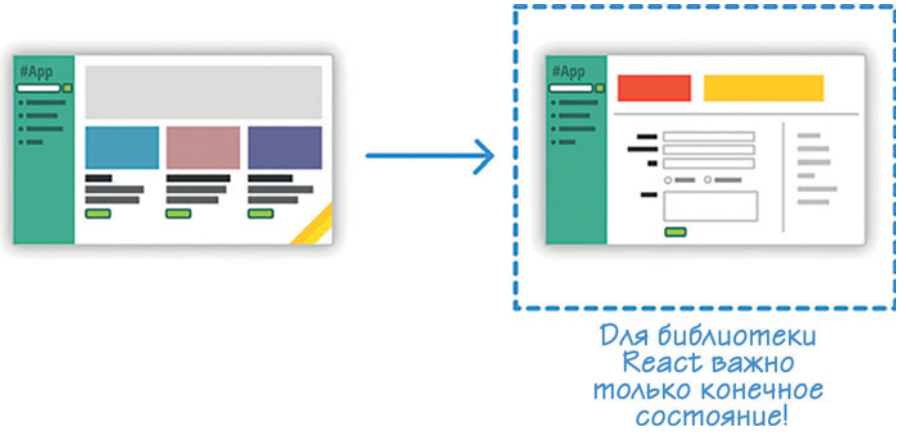


Рис. 1.7. Конечное состояние пользовательского интерфейса — это единственное, что имеет значение в React

Библиотека React заботится обо всем остальном. Она определяет, что должно быть сделано для правильного представления пользовательского интерфейса, поэтому все, что касается управления его состоянием, больше не является вашей заботой.

Быстрое манипулирование DOM

Поскольку изменение DOM происходит очень медленно, при использовании библиотеки React вы никогда не будете работать непосредственно с DOM. Вместо этого вы будете изменять хранящуюся в памяти *виртуальную* модель DOM (похожую на то, что изображено на рис. 1.8).

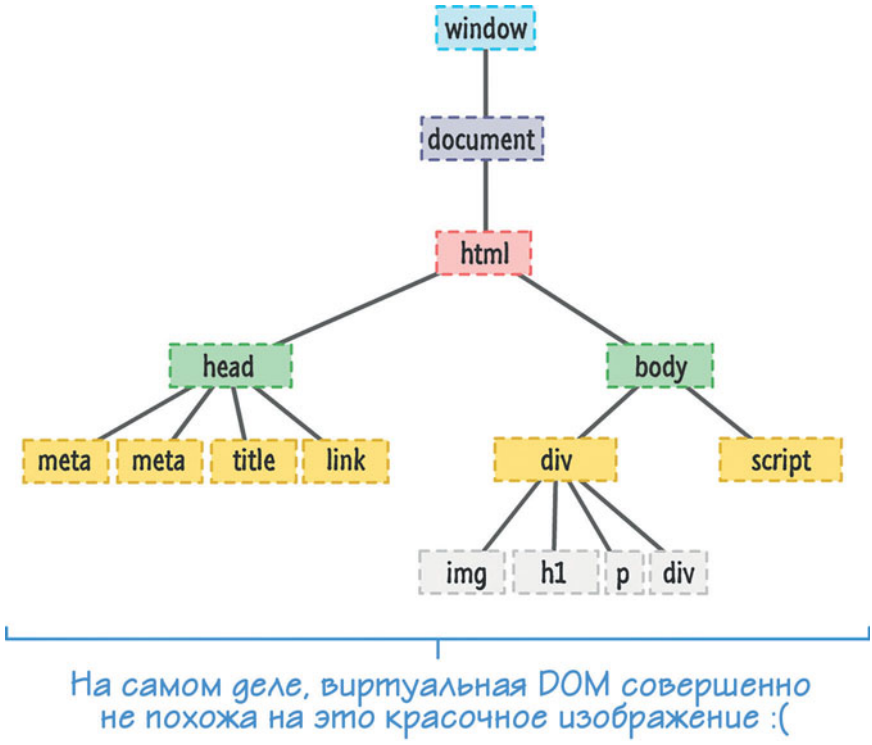


Рис. 1.8. Представьте хранящуюся в памяти виртуальную DOM, похожую на это

Манипулирование этой виртуальной DOM происходит чрезвычайно быстро, и в нужное время библиотека React обновляет реальную DOM. Для этого она сравнивает виртуальную DOM с реальной, выявляя критически важные преобразования, и вносит минимальные изменения, необходимые для поддержания актуальности DOM, в рамках процесса, называемого **сверкой** (reconciliation).

API для создания компонуемых пользовательских интерфейсов

Вместо того чтобы работать с визуальными элементами приложения как с единым монолитным фрагментом, библиотека React побуждает вас разбивать эти визуальные элементы на более мелкие компоненты (см. рис. 1.9).

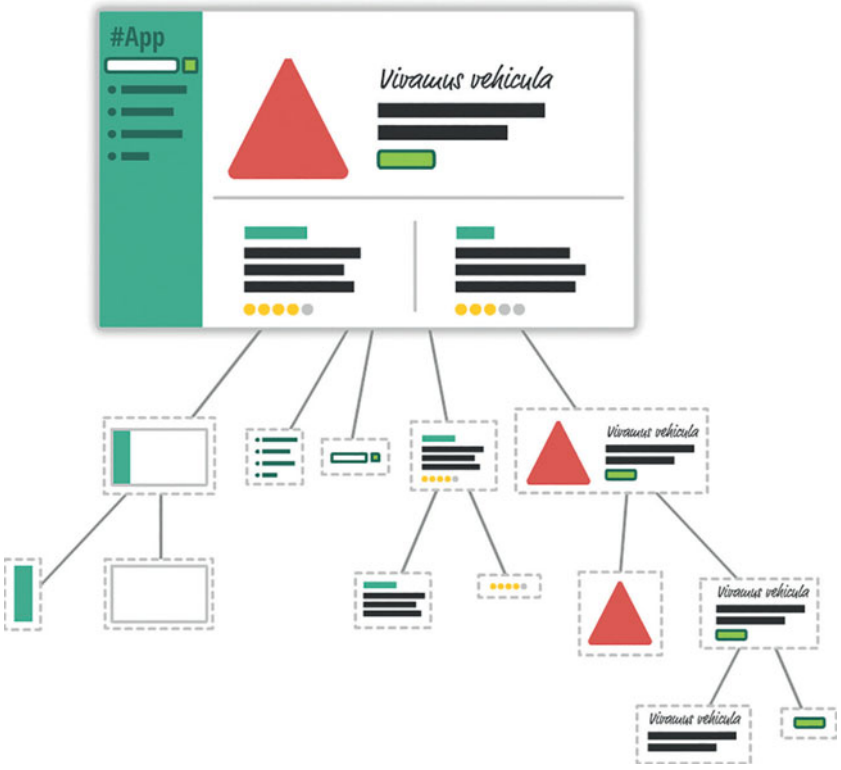


Рис. 1.9. Пример того, как визуальные элементы приложения могут быть разбиты на более мелкие фрагменты

Как и везде, в программировании следует работать с модульными, компактными и автономными элементами. Библиотека React позволяет применить эту проверенную временем идею к разработке пользовательских интерфейсов. Многие из базовых API библиотеки React направлены на облегчение процесса создания небольших визуальных компонентов, которые потом можно объединить с другими визуальными компонентами и создать более крупные и более сложные визуальные компоненты. Этот подход напоминает русские матрешки, изображенные на рис. 1.10.

Рис. 1.10. Матрешки

Это один из основных способов, с помощью которых библиотека React упрощает (и изменяет) подход к созданию визуальных элементов для веб-приложений.

Визуальные элементы, целиком определенные в коде JavaScript

Как бы безумно и возмутительно ни звучало то, что написано далее, выслушайте меня. Помимо странного синтаксиса, шаблоны HTML имеют другой серьезный недостаток: они ограничивают спектр действий, которые вы можете совершить внутри них, и речь идет не только об отображении данных. Например, если вы хотите выбрать для отображения фрагмент пользовательского интерфейса на основании определенного условия, вам придется написать код JavaScript где-то в своем приложении или использовать

какую-нибудь странную, специфическую для конкретного фреймворка шаблонную команду.

Например, вот как выглядит условная инструкция внутри шаблона EmberJS:

```
{{#if person}}
  С возвращением, {{person.firstName}} {{person.lastName}}!
{{else}}
  Пожалуйста, авторизуйся.
{{/if}}
```

Библиотека React делает кое-что замечательное. Благодаря тому, что пользовательский интерфейс целиком определен в коде JavaScript, вы можете использовать широкие функциональные возможности этого языка для выполнения различных действий внутри шаблонов. Вы ограничены только возможностями JavaScript, а не функциями фреймворка для шаблонов.

Думая о визуальных элементах, полностью определенных в коде JavaScript, вы, вероятно, представляете себе множество кавычек, escape-символов и вызовов `createElement`. Не волнуйтесь. Библиотека React позволяет (при необходимости) определить визуальные элементы с помощью HTML-подобного синтаксиса **JSX**, который сосуществует с кодом JavaScript. Вместо написания кода для определения пользовательского интерфейса вы лишь создаете разметку:

```
ReactDOM.render(
  <div>
    <h1>Бэтмен</h1>
    <h1>Терминатор</h1>
    <h1>Николас Кейдж</h1>
    <h1>Нео</h1>
  </div>,
  destination
);
```

В JavaScript этот же код выглядел бы так:

```
ReactDOM.render(React.createElement(
  "div",
  null,
  React.createElement(
    "h1",
    null,
    "Бэтмен"
  ),
  React.createElement(
    "h1",
    null,
    "Терминатор"
  ),
  React.createElement(
    "h1",
    null,
    "Николас Кейдж"
  ),
  React.createElement(
    "h1",
    null,
    "Нео"
  )
), destination);
```

К счастью, благодаря JSX вы можете легко определить свои визуальные элементы, используя знакомый синтаксис и не отказываясь при этом от мощи и гибкости языка JavaScript.

А лучше всего то, что в React визуальные элементы и JavaScript часто находятся в одном и том же месте. Вам больше не нужно переключаться между несколькими файлами, чтобы определить внешний вид и поведение одного визуального компонента. Это пример правильной работы с шаблонами.

Только V в архитектуре MVC

Мы почти закончили! React не является полноценным фреймворком, имеющим представление о том, как должны себя вести компоненты приложения. Вместо этого библиотека React в основном работает в слое представления, занимаясь поддержанием актуальности визуальных элементов. Это означает, что вы можете использовать все, что угодно, при работе над M (моделью) и C (контроллером) своей архитектуры MVC (Model-View-Controller, Модель-Представление-Контроллер). Такая гибкость позволяет вам выбирать технологии, с которыми вы уже знакомы, и делает библиотеку React полезной не только для вновь созданных вами веб-приложений, но и для существующих приложений, которые вы желаете улучшить, не прибегая к удалению и рефакторингу кода.

Заключение

Среди новых веб-фреймворков и библиотек React является безусловным успехом. Она не только решает самые распространенные проблемы разработчиков одностраничных приложений, но и предлагает дополнительные техники, которые *значительно* упрощают создание визуальных элементов для одностраничных приложений. С момента своего выхода в 2013 году библиотека React нашла применение в таких популярных веб-сайтах и приложениях, как Facebook, Instagram, BBC, Khan Academy, PayPal, Reddit, The New York Times и Yahoo!.

В этой главе мы узнали о библиотеке React и о том, что она делает. В следующих главах мы рассмотрим все это подробнее, а также обсудим технические детали, которые помогут вам успешно использовать React в собственных проектах. Никуда не уходите.

Глава 2
Создание первого React-приложения

Из предыдущей главы вы узнали о предыстории библиотеки React и о ее способности усовершенствовать самые сложные пользовательские интерфейсы. Несмотря на все преимущества React, начать работать с библиотекой не так просто. Она имеет крутую кривую обучения, и путь ее освоения заполнен множеством мелких и крупных препятствий (рис. 2.1).

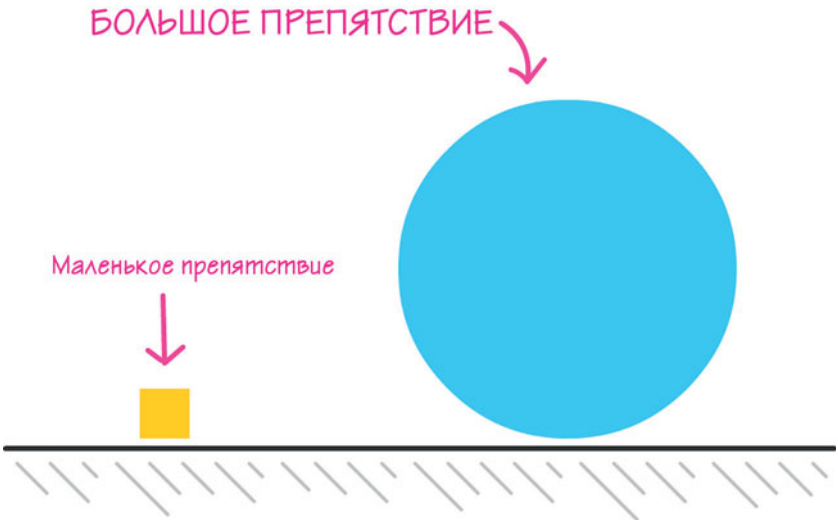


Рис. 2.1. Препятствия бывают большими и маленькими

В этой главе мы начнем с самого начала и создадим с помощью библиотеки React простое приложение. Вы столкнетесь с некоторыми из препятствий, а некоторые вам удастся обойти… на этот раз.

К концу этой главы вы не только создадите приложение, которым можно похвастаться друзьям и близким, но и прекрасно подготовитесь к более глубокому изучению библиотеки React в будущих главах.

Работа с JSX

Прежде чем приступить к созданию приложения, нам следует обратить внимание на один важный нюанс. React не похожа на прежние библиотеки JavaScript. Ее не слишком радует, когда вы ссылаетесь на написанный для нее код, используя элемент `script`. Эта раздражающая особенность влияет на процесс создания React-приложений.

Как вы знаете, веб-приложения (и все остальное, что отображается в браузере) состоят из кода на языках HTML, CSS и JavaScript (рис. 2.2).

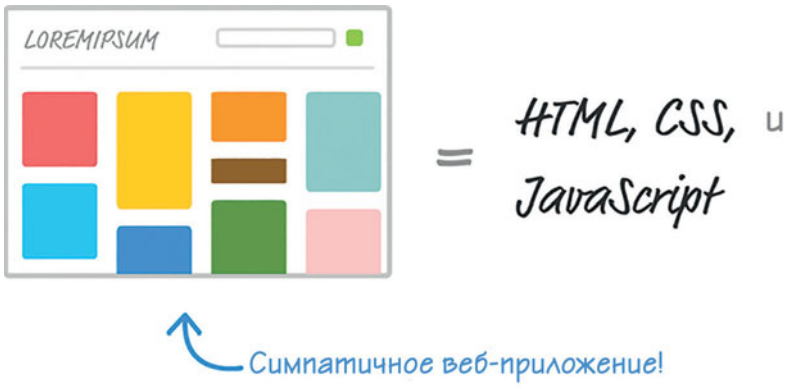


Рис. 2.2. Веб-приложения состоят из HTML, CSS и JavaScript

Неважно, создано ли веб-приложение с помощью React или другой библиотеки, вроде Angular, Knockout или jQuery. Конечный результат должен представлять собой комбинацию кода на языках HTML, CSS и JavaScript; в противном случае браузер не будет знать, что с ним делать.

Тут и проявляется особенность React. *Помимо обычного HTML, CSS и JavaScript основная часть вашего React-кода будет написана с использованием синтаксиса JSX*. Как я уже упоминал в главе 1,

JSX — это язык, который позволяет легко смешивать код JavaScript с HTML-подобными тегами для определения элементов пользовательского интерфейса (UI) и их функциональности. Это звучит здорово (и чуть позже вы увидите JSX в действии), однако есть небольшая проблема. Браузер не знает, что делать с JSX.

Чтобы создать веб-приложение с помощью библиотеки React, нам требуется способ преобразования JSX в старый добрый JavaScript, который браузер способен понять (рис. 2.3).

Рис. 2.3. JSX необходимо преобразовать в код, который понимает браузер

Если мы этого не сделаем, React-приложение не заработает. К счастью, у нас есть два решения этой проблемы:

1. **Настройте среду разработки с Node и несколькими инструментами для сборки приложений.** В этой среде после каждой сборки приложения весь код JSX автоматически преобразуется в JS и помещается на диск, чтобы можно было ссылаться на него, как на обычный файл JavaScript.

2. **Позвольте своему браузеру автоматически преобразовывать JSX в JavaScript во время выполнения.** Вы задаете код JSX непосредственно, как любой другой фрагмент JavaScript, а браузер заботится обо всем остальном.

Оба эти решения приемлемы, однако давайте разберем, какое влияние оказывает каждое из них.

Первое решение, несмотря на изначальную сложность и временные затраты, представляет собой современный способ веб-разработки. Помимо компиляции (точнее, транспиляции) JSX в JS, этот подход позволяет вам воспользоваться преимуществами модулей, более совершенных инструментов сборки и множеством других функций, облегчающих процесс создания сложных веб-приложений.

Второе решение — это быстрый и прямой путь, на котором вы изначально тратите больше времени на написание кода и меньше времени проводите в среде разработки. Для его использования вам достаточно сослаться на файл сценария. Этот файл сценария позаботится о преобразовании JSX в JS при загрузке страницы, и React-приложение оживет без особых действий в среде разработки с вашей стороны.

Для знакомства с React мы используем второе решение. Но почему мы выбираем второе решение не всегда? Дело в том, что процесс перевода JSX в JS снижает производительность браузера. Это вполне приемлемо в процессе изучения библиотеки React, но *недопустимо* при развертывании приложения для реального использования. Поэтому, когда вы освоитесь с React, мы вернемся к рассмотрению первого решения и обсудим процесс настройки среды разработки.

Использование React

В предыдущем разделе мы рассмотрели два способа, гарантирующих, что React-приложение окажется понятным для браузера. В этом разделе мы проверим упомянутые способы на практике. В качестве отправной точки мы используем пустую HTML-страницу.

Создайте HTML-документ со следующим содержимым.

```
<! DOCTYPE html>
<html>

<head>
  <meta charset="utf-8">
```

```
    <title>React! React! React!</title>
  </head>

  <body>
    <script>

    </script>
  </body>

  </html>
```

Эта страница не содержит ничего интересного, но мы исправим это, добавив ссылку на библиотеку React. Сразу под заголовком добавьте следующие две строки:

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

Эти две строки подключают как основную библиотеку React, так и все, что ей необходимо для работы с DOM. Без них вам не удастся создать React-приложение.

Это не все. Необходимо сослаться еще на одну библиотеку. Под этими двумя элементами `script` добавьте следующую строку:

```
<script src="https://unpkg.com/babel-standalone@6.15.0/
babel.min.js"></script>
```

Здесь вы добавляете ссылку на компилятор для JavaScript Babel (**babeljs.io**). Этот компилятор делает много интересного, однако для нас в первую очередь важна его способность превращать код JSX в JavaScript.

На данном этапе HTML-страница должна выглядеть следующим образом:

```
<! DOCTYPE html>
<html>

<head>
  <meta charset="utf-8">
  <title>React! React! React!</title>
  <script src="https://unpkg.com/react@16/umd/react.
      development.js"></script>
  <script src="https://unpkg.com/react-dom@16/umd/
      react-dom.development.js"></script>
  <script src="https://unpkg.com/babel-standalone@6.15.0/
      babel.min.js"></script>
</head>

<body>
  <script>

  </script>
</body>

</html>
```

Если вы просмотрите свою страницу прямо сейчас, то увидите — она по-прежнему пуста, на ней ничего не происходит. Это нормально. Далее мы это исправим.

Отображение имени

Сейчас вы используете библиотеку React, чтобы отобразить на экране свое имя. Для этого вы примените метод под названием `render`. Внутри пустого элемента `script` в теле страницы добавьте следующий код:

```
ReactDOM.render(
  <h1>Шерлок Холмс</h1>,
```

```
    document.body
);
```

Не беспокойтесь, если пока все это кажется бессмысленным. Сначала нам нужно подготовить содержимое для отображения на экране, а со всем остальным мы разберемся потом. Прежде чем просмотреть страницу и оценить результат, вам нужно обозначить этот блок сценария как нечто, с чем может работать компилятор Babel. Для этого вы добавляете к элементу `script` атрибут `type` со значением `text/babel`:

```
<script type="text/babel">
  ReactDOM.render(
    <h1>Шерлок Холмс</h1>,
    document.body
  );
</script>
```

После внесения этого изменения просмотрите страницу в своем браузере. Вы увидите слова «Шерлок Холмс», написанные гигантскими буквами (рис. 2.4).

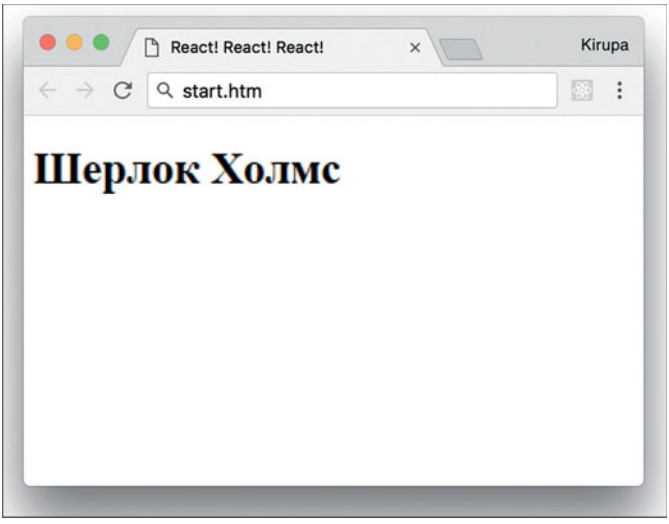


Рис. 2.4. В браузере должны отобразиться слова «Шерлок Холмс»

Поздравляю! Вы только что создали React-приложение.

Конечно, оно не такое уж впечатляющее. Кроме того, думаю, вас зовут не Шерлок Холмс. Однако, несмотря на примитивность, это приложение познакомило вас с одним из самых распространенных методов, которые часто используют при работе с React, — `ReactDOM.render`.

Метод `render` принимает два аргумента:

1. HTML-подобные элементы (JSX), которые необходимо вывести.

2. Место в DOM, где React должна отобразить JSX.

Вот как выглядит метод `render`:

```
ReactDOM.render(
  <h1>Шерлок Холмс</h1>,
  document.body
);
```

Первым аргументом является текст `Шерлок Холмс`, заключенный в элементе `h1`. Этот HTML-подобный синтаксис внутри кода JavaScript и есть JSX. Чуть позже мы рассмотрим его более подробно, а пока мне следует предупредить вас: *все настолько безумно, насколько кажется*. Всякий раз, когда я вижу скобки и слеши в коде JavaScript, что-то во мне обрывается от предчувствия того, что мне придется возиться с экранированием строк и кавычками. С JSX этого не нужно. Вы лишь помещаете свой HTML-подобный контент, как в вышеприведенном примере. И, словно по волшебству, все работает.

Вторым аргументом является `document.body`. В этом аргументе нет ничего странного или причудливого. Он лишь указывает конкретное место в DOM, где окажется преобразованная из JSX разметка. В примере при выполнении метода `render` элемент `h1` (и все его содержимое) помещается в элемент `body` документа.

Цель данного упражнения заключалась в том, чтобы отобразить на экране не *любое* имя, а *ваше* имя. Измените свой код

соответствующим образом. В моем случае метод `render` будет выглядеть следующим образом:

```
ReactDOM.render(
  <h1>Бэтмен</h1>,
  document.body
);
```

Ну, он выглядел бы так, если бы меня звали Бэтмен! В любом случае, если сейчас вы просмотрите свою страницу в браузере, то вместо слов «Шерлок Холмс» вы увидите свое имя.

Пока ничего нового

JavaScript выглядит иначе благодаря JSX, однако в итоге браузер имеет дело с чистым кодом на языке HTML, CSS и JavaScript. Чтобы убедиться в этом, мы немного изменим поведение и внешний вид приложения.

Смена места назначения

Сначала мы изменим место для вывода кода JSX. Использование JavaScript для помещения контента непосредственно в элемент `body` — это не лучшая идея. Многое может пойти не так, особенно если вы собираетесь использовать React наряду с другими библиотеками и фреймворками JS. Рекомендуется создать отдельный элемент, который будет рассматриваться как новый корневой элемент. Этот элемент используем в качестве места назначения для метода `render`. Вернитесь к коду HTML и добавьте элемент `div` с идентификатором `container`:

```
<body>
  <div id="container"></div>
  <script type="text/babel">
    ReactDOM.render(
```

```
        <h1>Бэтмен</h1>,
        document.body
      );
    </script>
  </body>
```

После определения **контейнера** `div` изменим метод `render`, чтобы использовать его вместо `document.body`. Вот один из способов это сделать:

```
ReactDOM.render(
  <h1>Бэтмен</h1>,
  document.querySelector("#container")
);
```

Другой способ заключается в добавлении переменной за пределами самого метода `render`:

```
var destination = document.querySelector("#container");

ReactDOM.render(
  <h1>Бэтмен</h1>,
  destination
);
```

Обратите внимание на то, что в переменной `destination` хранится ссылка на элемент-контейнер DOM. Внутри метода `render` вы ссылаетесь на переменную `destination` вместо использования синтаксиса для поиска элемента в качестве части самого аргумента. Причина этого проста: я хочу показать вам, что вы по-прежнему пишете код JavaScript и что `render` представляет собой еще один старый скучный метод, принимающий два аргумента.

Добавление стилей

Настало время внести последнее изменение. В настоящее время к именам применяется стиль `h1`, используемый браузером по умолчанию. Это ужасно, поэтому исправим это, добавив CSS. Внутри элемента `head` добавим блок `style` со следующим кодом CSS:

```
<style>
  #container {
    padding: 50px;
    background-color: #EEE;
  }
  #container h1 {
    font-size: 144px;
    font-family: sans-serif;
    color: #0080A8;
  }
</style>
```

После внесения всех изменений просмотрите свою страницу в браузере. Обратите внимание на то, что текст выглядит лучше, чем раньше, когда к нему применялся только стиль браузера по умолчанию (рис. 2.5).

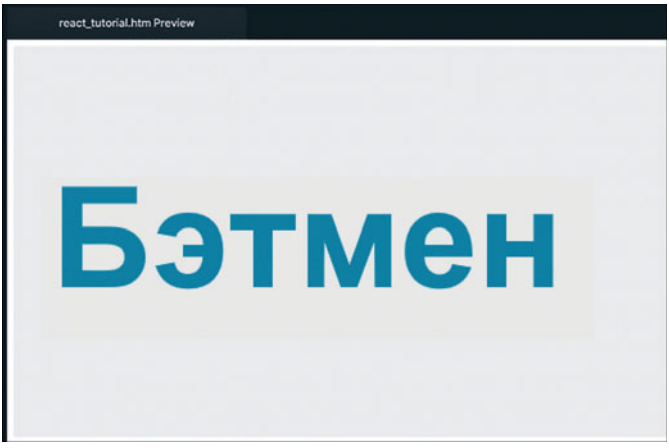


Рис. 2.5. Результат добавления CSS

Это работает, потому что после выполнения всего кода React DOM-элемент `body` содержит элемент `container` с находящимся внутри него элементом `h1`. Неважно, что элемент `h1` полностью определен в коде JavaScript в этом JSX-синтаксисе или что код CSS был определен вне метода `render`. Конечный результат работы React-приложения по-прежнему будет на 100% состоять из HTML, CSS и JavaScript. Взглянув на этот преобразованный JavaScript, мы бы увидели примерно следующее:

```
<! DOCTYPE html>
<html>

<head>
  <meta charset="utf-8">
  <title>React! React! React!</title>
<script src="https://unpkg.com/react@16/umd/react.
development.js"></script>
<script src="https://unpkg.com/react-dom@16/umd/react-dom.
development.js"></script>
<script src="https://unpkg.com/babel-standalone@6.15.0/
babel.min.js"></script>

  <style>
    #container {
    padding: 50px;
    background-color: #EEE;
    }
    #container h1 {
      font-size: 144px;
      font-family: sans-serif;
      color: #0080A8;
    }
  </style>
</head>
```

```
<body>
  <div id="container"></div>
<script type="text/babel">
    var destination = document.querySelector("#container");

    ReactDOM.render(React.createElement(
      "h1",
      null,
      "Бэтмен"
    ), destination);
</script>
</body>

</html>
```

Обратите внимание на то, что здесь нет никаких следов React-подобного кода.

Заключение

В этой главе мы познакомились с процессом создания React-приложений. Важно понимать, что React отличается от других библиотек, поскольку использует совершенно новый язык под названием JSX для определения внешнего вида визуальных элементов. Вы получили небольшое представление об этом, когда мы определили элемент `h1` внутри метода `render`.

Влияние JSX не ограничивается только способом определения элементов пользовательского интерфейса. Этот синтаксис также влияет на то, как вы создаете свое приложение в целом. Поскольку браузер не понимает JSX в его обычном виде, вам необходим промежуточный этап для преобразования этого JSX в JavaScript. Один из подходов состоит в том, чтобы заставить приложение генерировать транспилированный JavaScript-код, соответствующий исходному JSX. Другой подход (который мы использовали в этой главе) предполагает использование библиотеки Babel для преобразования

JSX в JavaScript в самом браузере. Несмотря на то, что из-за снижения производительности этот подход не рекомендуется применять в реальных приложениях, при знакомстве с React он может оказаться очень удобным.

В следующих главах мы подробнее изучим синтаксис JSX и выйдем за рамки метода `render`, рассмотрев все важные аспекты работы библиотеки React.

> Примечание: Если у вас возникнут какие-либо проблемы, задайте вопрос!
>
> Если у вас есть какие-либо вопросы или код не работает ожидаемым образом, не стесняйтесь задать вопрос! Опубликуйте его на форуме **forum.kirupa.com** и получите помощь от самых дружелюбных и знающих людей в Интернете!

Глава 3
Компоненты React

Компоненты — это одна из тех частей, которые заставляют библиотеку React, ну, реагировать! Они являются одним из основных способов определения визуальных элементов и взаимодействий, с которыми имеют дело пользователи вашего приложения. Например, на рис. 3.1 показано, как выглядит готовое приложение.

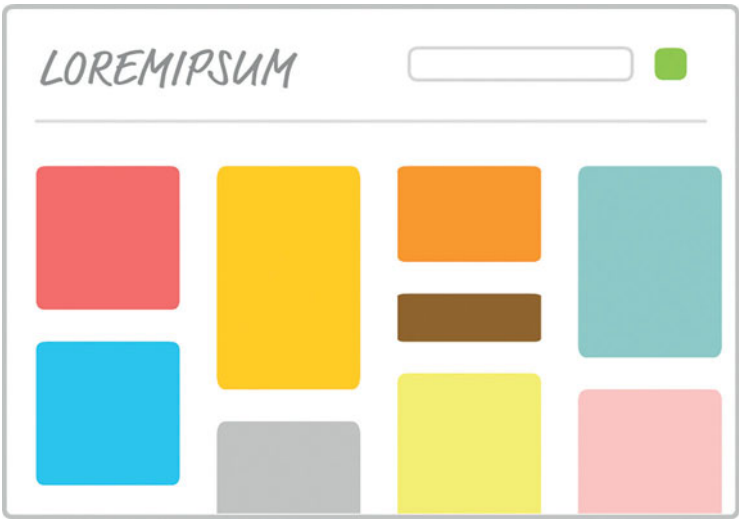


Рис. 3.1. Готовое приложение

Это готовое приложение. В процессе разработки, оцениваемое через призму проекта React, оно может показаться менее симпатичным. Почти каждая часть визуальных элементов этого приложения будет находиться внутри автономного модуля, называемого **компонентом**. Чтобы получить представление о том, что значит «почти каждая», посмотрите на рис. 3.2.

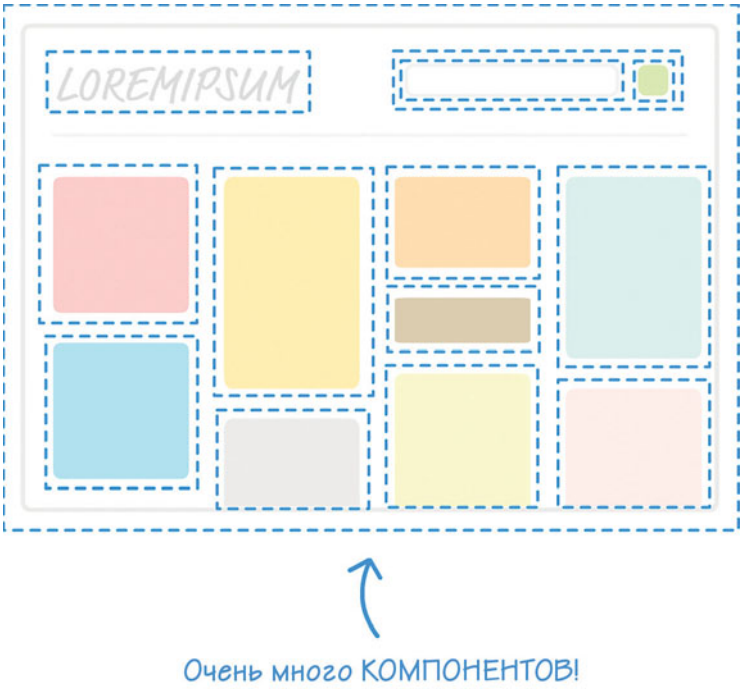


Рис. 3.2. Схематичное представление компонентов приложения

Пунктирными линиями обведены отдельные компоненты, которые отвечают за внешний вид и любые взаимодействия конкретных элементов. Не пугайтесь. Несмотря на видимую сложность, вы во всем разберетесь, немного поэкспериментировав с компонентами и их замечательными функциями.

Краткий обзор функций

В JavaScript есть **функции**, которые позволяют сделать код немного более кратким и допускающим многократное использование. Для рассмотрения функций есть причина, поэтому пусть вас это не раздражает! Концептуально функции имеют много общего с компонентами React, а чтобы понять принцип работы компонентов, проще всего сначала познакомиться с функциями.

В ужасном мире, в котором не существует функций, вы могли бы использовать вот такой код:

```
var speed = 10;
var time = 5;
alert(speed * time);

var speed1 = 85;
var time1 = 1.5;
alert(speed1 * time1);

var speed2 = 12;
var time2 = 9;
alert(speed2 * time2);

var speed3 = 42;
var time3 = 21;
alert(speed3 * time3);
```

В прекрасном мире, где функции существуют, вы можете сконцентрировать весь этот повторяющийся текст в нечто более простое, например:

```
function getDistance(speed, time) {
  var result = speed * time;
  alert(result);
}
```

Функция `getDistance` удаляет весь повторяющийся код, который вы видели ранее, и принимает скорость и время в качестве аргументов, позволяя вам повлиять на возвращаемый результат.

Для вызова этой функции достаточно сделать следующее:

```
getDistance(10, 5);
getDistance(85, 1.5);
getDistance(12, 9);
getDistance(42, 21);
```

Уже лучше, не правда ли? Однако преимущества функций этим не ограничиваются. Одни функции (например, `alert` внутри `getDistance`) могут вызывать другие в рамках своей работы. Взгляните на то, как функция `formatDistance` используется для изменения результата, возвращаемого функцией `getDistance`:

```
function formatDistance(distance) {
  return distance + " км";
}

function getDistance(speed, time) {
  var result = speed * time;
  alert(formatDistance(result));
}
```

Эта способность функций вызывать другие функции позволяет нам четко разделить их действия. Вам не нужно использовать одну монолитную функцию, которая выполняет все действия; вы можете разделить функциональность между многими функциями, которые специализируются на решении определенного типа задач.

А лучше всего то, что после внесения изменений в работу функций вам не нужно ничего делать, чтобы увидеть результаты этих изменений. Если сигнатура функции не поменялась, то любые ее вызовы волшебным образом сработают и автоматически учтут все изменения, внесенные в саму функцию.

Короче говоря, функции — это замечательно. Я это знаю. И вы это знаете. Вот почему написанный нами код переполнен ими.

Изменения во взаимодействии с пользовательским интерфейсом

Вряд ли кто-то не согласится с тем, что функции полезны. Они действительно позволяют разумно структурировать код приложений. Однако мы не всегда можем проявить такую заботу при разработке пользовательских интерфейсов. По различным техническим

и нетехническим причинам нам всегда приходилось допускать определенную небрежность в плане работы с элементами пользовательского интерфейса.

Это довольно спорное утверждение, поэтому позвольте мне пояснить, что я имею в виду, на нескольких примерах. Вернемся к методу `render`, который использовался в предыдущей главе:

```
var destination = document.querySelector("#container");

ReactDOM.render(
  <h1>Бэтмен</h1>,
  destination
);
```

На экране вы видите слово **Бэтмен**, написанное огромными буквами благодаря элементу `h1`. Давайте немного изменим ситуацию. Допустим, мы хотим вывести на экран имена еще нескольких супергероев. Для этого мы изменим метод `render` следующим образом:

```
var destination = document.querySelector("#container");

ReactDOM.render(
  <div>
    <h1>Бэтмен</h1>
    <h1>Терминатор</h1>
    <h1>Николас Кейдж</h1>
    <h1>Нео</h1>
  </div>,
  destination
);
```

Обратите внимание, что вы здесь видите. Мы используем контейнер `div`, содержащий четыре элемента `h1` с именами супергероев.

Итак, теперь у нас есть четыре элемента `h1`, каждый из которых содержит имя супергероя. Что, если мы хотим изменить элемент `h1`,

например, на `h3`? Мы можем вручную обновить все эти элементы следующим образом:

```
var destination = document.querySelector("#container");

ReactDOM.render(
  <div>
    <h3>Бэтмен</h3>
    <h3>Терминатор</h3>
    <h3>Николас Кейдж</h3>
    <h3>Нео</h3>
  </div>,
  destination
);
```

Если вы просмотрите страницу, то увидите, что ее содержимое выглядит простовато (рис. 3.3).

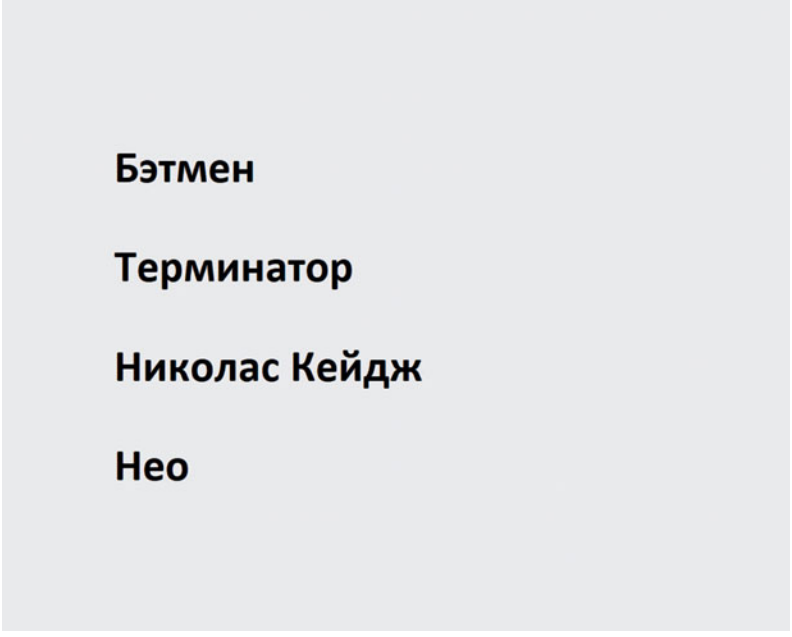


Рис. 3.3. Скучное оформление имен супергероев

Мы не собираемся сходить с ума по поводу стиля. Все, что нам нужно сделать, это выделить все имена курсивом, используя элемент `i`, поэтому давайте вручную обновим выводимый на экран контент:

```
var destination = document.querySelector("#container");

ReactDOM.render(
  <div>
    <h3><i>Бэтмен</i></h3>
    <h3><i>Терминатор</i></h3>
    <h3><i>Николас Кейдж</i></h3>
    <h3><i>Нео</i></h3>
  </div>,
  destination
);
```

Мы обошли все элементы `h3` и обернули их содержимое тегами элемента `i`. Вы уже заметили проблему? То, что мы делаем с пользовательским интерфейсом, ничем не отличается от следующего кода:

```
var speed = 10;
var time = 5;
alert(speed * time);

var speed1 = 85;
var time1 = 1.5;
alert(speed1 * time1);

var speed2 = 12;
var time2 = 9;
alert(speed2 * time2);

var speed3 = 42;
var time3 = 21;
alert(speed3 * time3);
```

Каждое изменение, которое мы хотим внести в элементы `h1` или `h3`, должно быть продублировано для каждого экземпляра. Что, если нам потребуется не просто изменить внешний вид элементов? Что, если мы захотим представить нечто более сложное, чем использованные примеры? То, что мы сейчас делаем, нельзя масштабировать; ручное обновление каждой копии того, что требуется изменить, занимает много времени. Кроме того, это скучно.

Вот сумасшедшая мысль: *Что, если бы все удивительные свойства функций можно было применить к определению визуальных элементов приложения*? Разве это не позволило бы решить проблему неэффективности, описанную в этом разделе? Как выясняется, ответ на это «что, если» и составляет суть библиотеки React. Пришло время познакомиться с **компонентом**.

Встречайте компонент React

Решение всех наших проблем (даже экзистенциальных) можно найти в компонентах React. *Компоненты React — это многократно используемые фрагменты JavaScript-кода, которые выводят HTML-элементы (благодаря JSX)*. Это звучит не особенно впечатляюще, однако по мере изучения и создания все более сложных компонентов вы оцените их мощь и преимущества.

Для начала создадим пару компонентов вместе. Чтобы повторять за мной, создайте чистый документ React:

```
<! DOCTYPE html>
<html>

<head>
  <meta charset="utf-8">
  <title>Компоненты React</title>
  <script src="https://unpkg.com/react@16/umd/react.
      development.js"></script>
  <script src="https://unpkg.com/react-dom@16/umd/react-dom.
      development.js"></script>
```

```
  <script src="https://unpkg.com/babel-standalone@6.15.0/
      babel.min.js"></script>
</head>

<body>
  <div id="container"></div>
  <script type="text/babel">

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

Пока на этой странице не происходит ничего интересного. Как и в предыдущей главе, эта страница довольно банальна, она содержит только ссылки на библиотеки React и Babel, а также элемент `div`, который имеет идентификатор `container`.

Создание компонента HelloWorld

Начнем с простого. Нам нужен компонент, позволяющий вывести на экран знаменитую фразу «Привет, мир!». Как мы уже знаем, при использовании только метода `ReactDOM.render` мы получим код, который выглядит следующим образом:

```
ReactDOM.render(
  <div>
    <p>Привет, мир!</p>
  </div>,
  document.querySelector("#container")
);
```

Давайте воссоздадим этот результат с помощью компонента. Библиотека React предусматривает несколько способов создания компонентов, однако мы воспользуемся для этого синтаксисом класса. Добавьте выделенный цветом код над существующим методом `render`:

```
class HelloWorld extends React.Component {

}

ReactDOM.render(
  <div>
    <p>Привет, мир!</p>
  </div>,
  document.querySelector("#container")
);
```

Если вы не знакомы с синтаксисом класса, изучите мое онлайн-руководство под названием Using Classes in JavaScript (**www.kirupa.com/javascript/classy_way_to_create_objects.htm**).

Вернемся к коду. Мы создали новый компонент `HelloWorld`. Это компонент, поскольку он расширяет класс `React.Component`. Если бы он этого не делал, то представлял бы собой пустой класс, который мало чем был бы полезен. Внутрь этого класса вы можете поместить всевозможные методы для дальнейшего определения функций компонента `HelloWorld`. Некоторые из определяемых вами методов являются особенными, и библиотека React использует их, чтобы помочь компонентам творить чудеса. Одним из таких незаменимых методов является `render`.

Измените компонент `HelloWorld`, добавив метод `render`, как показано далее:

```
class HelloWorld extends React.Component {
  render() {

  }
}
```

Подобно методу `render`, который мы рассмотрели раньше в качестве части `ReactDOM.render`, функция `render` внутри компонента

также отвечает за взаимодействие с кодом JSX. Давайте изменим функцию `render`, чтобы вывести на экран фразу **Привет, компонентизированный мир!**. Добавьте в код следующую выделенную цветом строку:

```
class HelloWorld extends React.Component {
  render() {
    return <p>Привет, компонентизированный мир!</p>
  }
}
```

Вы приказали функции `render` возвратить код JSX, который представляет собой текст **Привет, компонентизированный мир!**. Остается только использовать этот компонент. Вы сделаете это после того, как определите его путем **вызова**. Здесь мы вызываем его из уже знакомого нам метода `ReactDOM.render`.

Способ вызова метода из компонента имеет некоторые особенности. Замените первый аргумент `ReactDOM.render` следующим содержимым:

```
ReactDOM.render(
  <HelloWorld/>,
  document.querySelector("#container")
);
```

Это не опечатка! Код JSX, который мы используем для вызова компонента `HelloWorld`, представляет собой HTML-подобный код `<HelloWorld/>`. Если вы просмотрите страницу в своем браузере, то увидите на экране текст **Привет, компонентизированный мир!**. Если вы задерживали дыхание в ожидании результата, можете расслабиться.

Если вам трудно расслабиться после того, как вы увидели синтаксис, использованный для вызова `HelloWorld`, посмотрите на круг на рис. 3.4 в течение нескольких секунд.

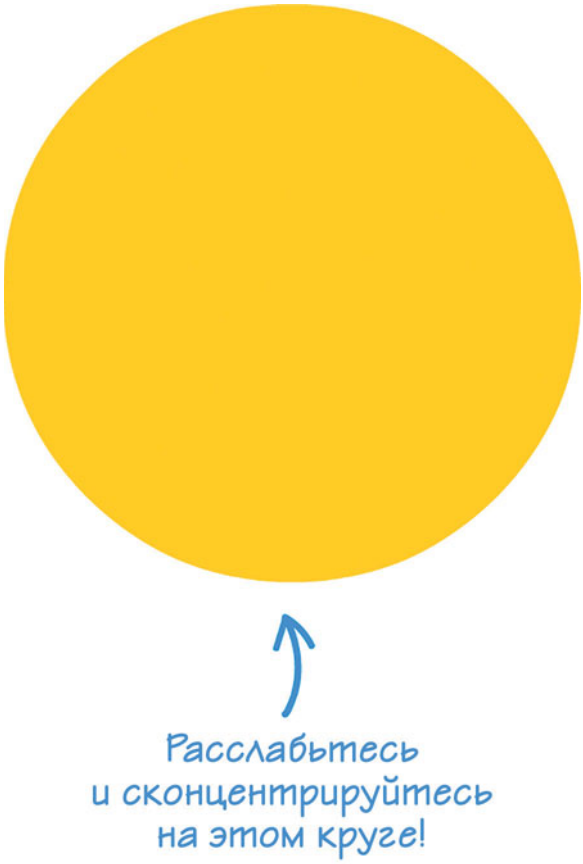


Рис. 3.4. Просто небольшое беззаботное развлечение!

Возвращаемся в реальность. То, что мы делали до сих пор, может показаться безумием, однако думайте о своем компоненте `<HelloWorld/>` как о классном новом HTML-элементе, функциональность которого находится под вашим полным контролем. Это означает, что вы можете выполнять над ним всевозможные HTML-подобные операции.

Например, измените метод `ReactDOM.render` так, чтобы он выглядел следующим образом:

```
ReactDOM.render(
  <div>
```

```
      <HelloWorld/>
    </div>,
    document.querySelector("#container")
);
```

Мы поместили вызов компонента `HelloWorld` в элемент `div`, и если вы сейчас просмотрите страницу в своем браузере, то убедитесь в том, что все работает. Давайте продвинемся еще на шаг вперед! Вместо одного вызова `HelloWorld` сделаем несколько. Измените метод `ReactDOM.render`, чтобы он выглядел следующим образом:

```
ReactDOM.render(
  <div>
    <HelloWorld/>
    <HelloWorld/>
    <HelloWorld/>
    <HelloWorld/>
    <HelloWorld/>
    <HelloWorld/>
  </div>,
  document.querySelector("#container")
);
```

Теперь вы увидите несколько текстовых строк **Привет, компонентизированный мир!**. Давайте сделаем еще кое-что, прежде чем перейти к более интересным задачам. Вернитесь к объявлению компонента `HelloWorld` и измените текст на более привычную фразу **Привет, мир!**:

```
class HelloWorld extends React.Component {
  render() {
    return <p>Привет, мир!</p>
  }
}
```

Внесите это изменение, а затем просмотрите свою страницу. На этот раз все указанные ранее вызовы `HelloWorld` выведут на экран фразу **Привет, мир!**. Вручную изменять каждый вызов `HelloWorld` не пришлось, — это хорошо!

Указание свойств

В настоящее время наш компонент выполняет только одну функцию. Он выводит фразу **Привет, мир!** на экран — и все! Это эквивалент функции JavaScript, которая выглядит так:

```
function getDistance() {
  alert("42км");
}
```

За исключением одного особого случая, данная функция JavaScript не кажется очень уж полезной, не так ли? Чтобы сделать эту функцию полезнее, нам нужно ее модифицировать, чтобы она принимала аргументы:

```
function getDistance(speed, time) {
  var result = speed * time;
  alert(result);
}
```

Теперь эту функцию можно использовать в более широком спектре ситуаций, а не только в тех, где выходом является **42км**.

Что-то подобное относится и к вашим компонентам. Как и в случае с функциями, вы можете передать аргументы, изменяющие то, что делает компонент. Здесь следует разобраться с терминами. То, что мы называем **аргументами** в мире функций, в мире компонентов называется **свойствами**. Давайте посмотрим на эти свойства в действии!

Теперь вам предстоит изменить компонент `HelloWorld` так, чтобы вы могли использовать в качестве приветствия что-то, кроме

общего слова **мир**. Например, представьте, что вы можете использовать слово **Боня** в качестве части вызова `HelloWorld`, чтобы увидеть на экране фразу **Привет, Боня!**.

Для добавления свойств к компоненту вам нужно следовать инструкции, состоящей из двух частей.

Часть первая: Обновление определения компонента

В настоящее время наш компонент `HelloWorld` жестко запрограммирован на выдачу фразы **Привет, мир!** в качестве части возвращаемого им значения. Сначала нам нужно изменить это поведение, заставив инструкцию `return` выводить на экран значение, переданное свойством. Зададим имя для свойства; в данном примере это будет `greetTarget`.

Чтобы указать значение `greetTarget` в качестве части нашего компонента, нам нужно внести следующее изменение:

```
class HelloWorld extends React.Component {
  render() {
    return <p>Привет, {this.props.greetTarget}!</p>
  }
}
```

Вы получаете доступ к свойству, ссылаясь на него с помощью свойства `this.props`, доступ к которому имеет каждый компонент. Обратите внимание на то, как указывается это свойство: оно заключается в фигурные скобки `{` и `}`. *В JSX, если вы хотите, чтобы нечто оценивалось как выражение, вам нужно заключить это в фигурные скобки*. Если вы этого не сделаете, то увидите на экране текст `this.props.greetTarget`.

Вторая часть: Изменение кода вызова компонента

После обновления определения компонента вам останется только передать значение свойства как часть кода вызова компонента. Для

этого нужно добавить атрибут с именем свойства, за которым следует значение, которое вы хотите передать. В примере в код вызова `HelloWorld` требуется добавить атрибут `greetTarget` и значение, которое вы хотите ему присвоить.

Измените код вызовов `HelloWorld` следующим образом:

```
ReactDOM.render(
  <div>
    <HelloWorld greetTarget="Бэтмен"/>
    <HelloWorld greetTarget="Терминатор"/>
    <HelloWorld greetTarget="Николас Кейдж"/>
    <HelloWorld greetTarget="Нео"/>
    <HelloWorld greetTarget="Боня"/>
    <HelloWorld greetTarget="Женщина-кошка"/>
  </div>,
  document.querySelector("#container")
);
```

Теперь каждый вызов `HelloWorld` имеет атрибут `greetTarget` и имя супергероя (или другого мифического существа), которого мы хотим поприветствовать. Если вы просмотрите эту страницу в браузере, то увидите, что эти приветствия успешно выводятся на экран.

Прежде чем двигаться дальше, необходимо упомянуть еще об одной важной вещи. Вы не ограничены только одним свойством в компоненте. У вас может быть любое количество свойств, и свойство `props` без проблем справится с любыми имеющимися запросами.

Работа с дочерними элементами

Ранее я упомянул о том, что компоненты (в JSX) очень похожи на обычные HTML-элементы. Вы могли убедиться в этом, поместив компонент в элемент `div` или указав атрибут и значение при задании свойств. *Вы можете использовать множество HTML-элементов, а компоненты могут иметь дочерние элементы.*

Это означает, что вы можете сделать что-то вроде этого:

```
<CleverComponent foo="bar">
  <p>Что-то!</p>
</CleverComponent>
```

Здесь мы используем компонент под названием `CleverComponent`, у которого есть дочерний элемент `p`. Из `CleverComponent` вы можете получить доступ к дочернему элементу `p` (и к любому другому дочернему элементу) с помощью свойства `children`, доступ к которому осуществляется с помощью `this.props.children`.

Чтобы разобраться во всем этом, рассмотрим другой простой пример. На этот раз у нас есть компонент `Buttonify`, который помещает дочерние элементы в элемент `button`. Этот компонент выглядит следующим образом:

```
class Buttonify extends React.Component {
  render() {
    return(
      <div>
      <button type={this.props.behavior}>{this.props.
              children}</button>
      </div>
    );
  }
}
```

Вы можете использовать этот компонент, вызвав его с помощью метода `ReactDOM.render`, как показано ниже:

```
ReactDOM.render(
  <div>
    <Buttonify behavior="submit">ОТПРАВИТЬ ДАННЫЕ</Buttonify>
  </div>,
  document.querySelector("#container")
);
```

При выполнении этого кода, учитывая то, как выглядел код JSX в методе `render` компонента `Buttonify`, вы увидите слова **ОТПРАВИТЬ ДАННЫЕ**, заключенные внутри элемента `button`. При использовании соответствующего стиля результат может оказаться смехотворно большим (рис. 3.5).

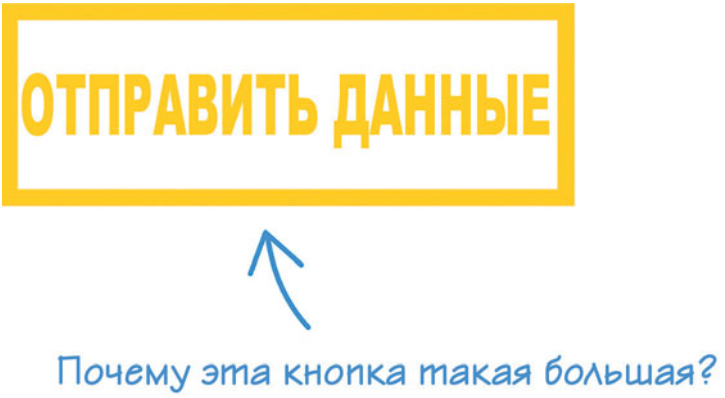


Рис. 3.5. Большая кнопка ОТПРАВИТЬ ДАННЫЕ

Обратите внимание на то, что в код JSX было добавлено свойство `behavior`. Оно позволяет нам указать атрибут `type` элемента `button`, и вы видите, что мы получаем к нему доступ с помощью свойства `this.props.behavior` в методе `render` определения компонента.

Доступ к дочерним элементам компонента не ограничивается тем, что вы только что увидели. Например, если дочерний элемент представляет собой простой текст, то свойство `this.props.children` возвратит строку. Если дочерний элемент — одиночный (как в нашем примере), то свойство `this.props.children` возвратит один компонент, который *не обернут* в массив. Нам нужно вызвать еще несколько вещей, однако, чтобы вас не утомлять, мы рассмотрим все эти подробности позднее при обсуждении более сложных примеров.

Заключение

Если вы хотите создать приложение с помощью библиотеки React, вы не сможете далеко продвинуться без использования компонентов. Создавать React-приложения без применения компонентов — это все равно, что создавать JavaScript-приложения без использования функций. Я не говорю, что это невозможно; это лишь один из примеров *Плохой идеи,* как в популярных мультипликационных зарисовках Animaniacs Good Idea/Bad Idea (**www.youtube.com/watch?v=2dJOIf4mdus**).

Если это остроумное видео не убедило вас в целесообразности знакомства с компонентами, я не знаю, что убедит. Быть может, одна из следующих глав, посвященная созданию сложных компонентов!

> Примечание: Если у вас возникнут какие-либо проблемы, задайте вопрос!
>
> Если у вас есть какие-либо вопросы или код не работает ожидаемым образом, не стесняйтесь задать вопрос! Опубликуйте свой вопрос на форуме **forum.kirupa.com** и получите помощь от самых дружелюбных и знающих людей в Интернете!

Глава 4
Стили в библиотеке React

На протяжении многих поколений человечество (и, вероятно, очень умные дельфины) оформляло HTML-контент, используя CSS. Все было замечательно. CSS предусматривает хорошее разделение содержимого и представления. Синтаксис селекторов обеспечивает большую гибкость при принятии решения о том, к каким элементам применять стиль, а к каким — нет. У нас даже не было причин ненавидеть *все это каскадирование*.

Только не говорите об этом React. Нельзя сказать, что эта библиотека активно сопротивляется CSS, она лишь имеет другое представление о стилизации контента. Как вы уже видели, одна из основных идей React заключается в том, чтобы сделать визуальные элементы приложения автономными и допускающими многоразовое использование. Вот почему элементы HTML и воздействующий на них код JavaScript находятся в одном блоке, называемом **компонентом**. Вы уже столкнулись с этим в предыдущей главе.

А как насчет внешнего вида HTML-элементов (то есть их стиля)? Где должна находиться информация о нем? Вы, вероятно, уже догадываетесь, к чему я веду. Элемент пользовательского интерфейса не может быть автономным, если его стиль определен где-то в другом месте. Вот почему библиотека React рекомендует вам хранить данные о внешнем виде элементов вместе с HTML и JavaScript. В нашем уроке вы узнаете все об этом таинственном (и, можно сказать, скандальном) подходе к стилизации контента. Конечно, мы также поговорим об использовании CSS. Мы можем применять оба подхода, несмотря на то, что по этому поводу думает React.

Отображение нескольких гласных

Чтобы разобраться со стилизацией контента React, давайте поработаем над (совершенно замечательным) кодом, который отображает на странице гласные буквы. Для начала вам понадобится пустая HTML-страница, которая будет содержать контент React. Создайте HTML-документ и добавьте в него следующее содержимое:

```
<! DOCTYPE html>
<html>

<head>
  <meta charset="utf-8">
  <title>Стили в React</title>
  <script src="https://unpkg.com/react@16/umd/react.
      development.js"></script>
  <script src="https://unpkg.com/react-dom@16/umd/react-dom.
      development.js"></script>
  <script src="https://unpkg.com/babel-standalone@6.15.0/
    babel.min.js"></script>

  <style>
    #container {
      padding: 50px;
      background-color: #FFF;
    }
  </style>
</head>

<body>
  <div id="container"></div>

</body>
</html>
```

Чтобы отобразить на странице гласные буквы, вам нужно добавить специфический код React. Сразу под **контейнером** `div` добавьте следующий код:

```
<script type="text/babel">
  var destination = document.querySelector("#container");

  class Letter extends React.Component {
    render() {
      return(
        <div>
          {this.props.children}
        </div>
      );
    }
}

  ReactDOM.render(
    <div>
      <Letter>А</Letter>
      <Letter>Е</Letter>
      <Letter>И</Letter>
      <Letter>О</Letter>
      <Letter>У</Letter>
    </div>,
    destination
  );
</script>
```

Благодаря тому, что вы узнали о компонентах раньше, здесь ничего не должно показаться загадочным. Вы создаете компонент под названием `Letter`, который отвечает за помещение гласных в элемент `div`.

Если вы просмотрите свою страницу, то увидите что-то скучное, вроде того, что изображено на рис. 4.1.

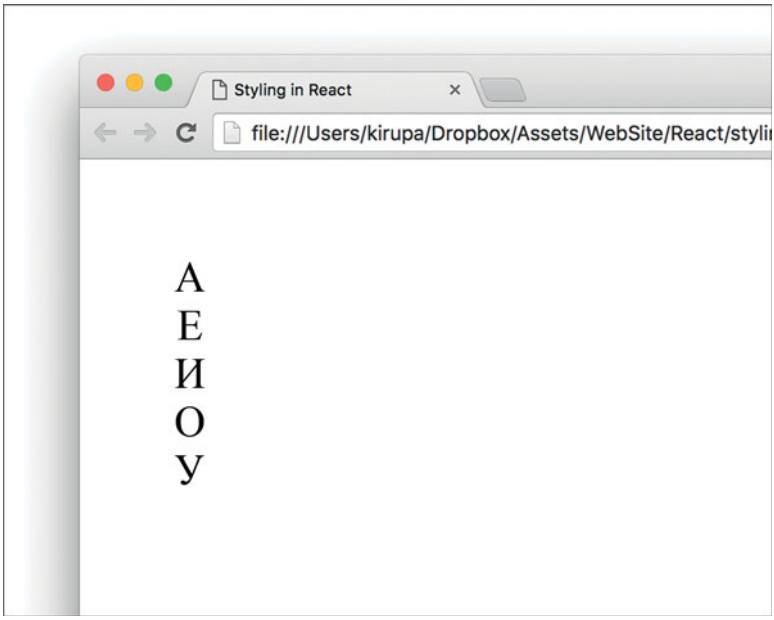


Рис. 4.1. Скучное содержимое экрана

Не беспокойтесь, сейчас вы сделаете содержимое экрана намного менее скучным. После внесения некоторых изменений вы увидите что-то более похожее на рис. 4.2.

Рис. 4.2. Содержимое экрана после применения стиля

Гласные будут отображены на желтом фоне, выровнены по горизонтали и отформатированы моноширинным шрифтом. Давайте посмотрим, как все это делается с помощью CSS и с использованием новомодного подхода React.

Стилизация контента React с помощью CSS

Использовать CSS для стилизации контента React на самом деле настолько просто, насколько это можно себе представить. Поскольку библиотека React в конечном итоге выдает обычные HTML-элементы, вы можете применить все свои навыки работы с CSS. Вам лишь нужно помнить о нескольких нюансах.

Разберитесь в сгенерированном HTML-коде

Прежде чем использовать CSS, вам нужно получить представление о том, как будет выглядеть HTML-код, сгенерированный React. Вы можете легко это сделать, посмотрев на код JSX, определенный внутри методов `render`. Родственный (предок) метод `render` основан на `ReactDOM` и выглядит следующим образом:

```
<div>
  <Letter>А</Letter>
  <Letter>Е</Letter>
  <Letter>И</Letter>
  <Letter>О</Letter>
  <Letter>У</Letter>
</div>
```

У нас есть несколько компонентов `Letter`, помещенных в элемент `div`. Пока ничего интересного. Метод `render` внутри компонента `Letter` тоже мало чем отличается:

```
<div>
  {this.props.children}
</div>
```

Как видите, каждая гласная заключена в отдельный набор тегов `div`. Если вы просмотрите этот пример в браузере, то увидите, что окончательная структура DOM для гласных будет выглядеть так, как показано на рис. 4.3.

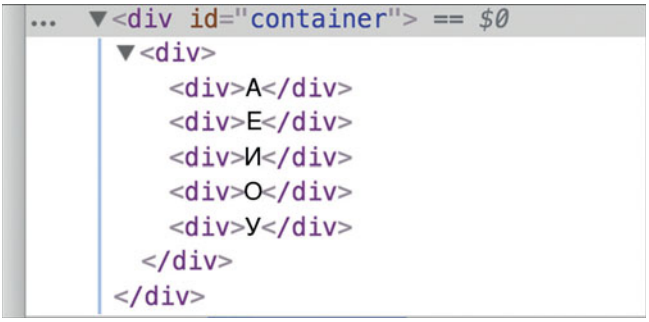


Рис. 4.3. Предварительный просмотр кода в браузере

В результате мы получили HTML-версию различных фрагментов JSX, виденных чуть ранее в методе `render`, с гласными, вложенными в несколько элементов `div`.

Примените стиль

Если вы поняли, как организованы HTML-элементы, к которым хотите применить стиль, считайте, самое сложное позади. Теперь настала очередь интересной и знакомой части работы, связанной с определением селекторов стилей и указанием свойств. Чтобы повлиять на внутренние элементы `div`, добавьте следующий код в элемент `style`:

```
div div div {
  padding: 10px;
  margin: 10px;
  background-color: #FFDE00;
  color: #333;
  display: inline-block;
  font-family: monospace;
  font-size: 32px;
  text-align: center;
}
```

Селектор `div div div` гарантирует, что стиль будет применен к нужным элементам. В результате гласные будут выглядеть так, как

мы задумали изначально. Учитывая сказанное, селектор стиля `div div div` выглядит немного странно, не правда ли? Он имеет слишком общий характер. В приложениях, содержащих более трех элементов `div` (что является обычным явлением), стиль может быть применен не к тем элементам. В таких случаях вам может понадобиться изменить создаваемый библиотекой React HTML-код, чтобы облегчить стилизацию контента.

Для этого мы назначим внутренним элементам `div` класс `letter`. Здесь мы сталкиваемся с отличием JSX от HTML. Добавьте в код выделенный цветом фрагмент:

```
class Letter extends React.Component {
  render() {
    return (
      <div className="letter">
      {this.props.children}
      </div>
    );
  }
}
```

Обратите внимание, что мы назначаем класс, используя атрибут `className` вместо атрибута `class`. Это связано с тем, что слово *class* является специальным ключевым словом в языке JavaScript. Если это пока ни о чем вам не говорит, не волнуйтесь; мы вернемся к этому позже.

После присвоения элементу `div` атрибута `className` со значением `letter`, осталось сделать только одну вещь. Измените селектор CSS для более точного нацеливания на элементы `div`:

```
.letter {
  padding: 10px;
  margin: 10px;
  background-color: #FFDE00;
  color: #333;
  display: inline-block;
```

```
    font-family: monospace;
    font-size: 32px;
    text-align: center;
}
```

Как видите, использование CSS — это вполне допустимый способ стилизации контента в React-приложениях. В следующем разделе мы рассмотрим способ стилизации, предпочитаемый библиотекой React.

Стилизация контента методом библиотеки React

Для библиотеки React предпочтительным является применение к контенту встроенных (в строку) стилей, то есть подход, который не предусматривает использование CSS. Поначалу это может показаться немного странным, однако такой подход облегчает повторное использование визуальных элементов. Цель состоит в том, чтобы превратить компоненты в небольшие черные ящики, где содержатся все данные, имеющие отношение к тому, как выглядит и функционирует пользовательский интерфейс. Давайте посмотрим, как это происходит.

В продолжение работы с начатым ранее примером удалите правило стиля `.letter`. Если вы просмотрите свой файл в браузере, то увидите, что гласные вернулись в свое исходное нестилизованное состояние. Вам также следует удалить объявление `className` из функции `render` компонента `Letter`. Нет смысла оставлять фрагменты разметки, которые вы не собираетесь использовать.

Теперь давайте вернем компонент `Letter` в исходное состояние:

```
class Letter extends React.Component {
  render() {
    return (
      <div>
        {this.props.children}
      </div>
    );
  }
}
```

Вы указываете стили внутри компонента путем определения объекта, содержащего CSS-свойства и их значения. После создания этого объекта вы назначаете его элементам JSX, которые хотите стилизовать, используя атрибут `style`. Все станет понятнее, когда вы выполните эти два действия самостоятельно, поэтому давайте применим описанный способ для стилизации компонента `Letter`.

Создание объекта стиля

Сначала определим объект, содержащий стили, которые мы хотим применить:

```
class Letter extends React.Component {
  render() {
    var letterStyle = {
      padding: 10,
      margin: 10,
      backgroundColor: "#FFDE00",
      color: "#333",
      display: "inline-block",
      fontFamily: "monospace",
      fontSize: 32,
      textAlign: "center"
    };

    return (
      <div>
        {this.props.children}
      </div>
    );
  }
}
```

У нас есть объект под названием `letterStyle`, в котором содержатся имена свойств CSS и их значения. Если вы никогда раньше

не определяли свойства CSS в JavaScript (путем установки `object.style`), то можете воспользоваться следующей простой формулой для их преобразования в нечто дружественное JavaScript:

Свойства CSS, состоящие из одного слова (например, `padding`, `margin` и `color`), остаются без изменения.

Свойства CSS, состоящие из нескольких слов, разделенных тире (например, `background-color`, `font-family` и `border-radius`), превращаются в одно слово, при этом тире удаляется, а первая буква второго слова становится заглавной. В нашем примере свойство `background-color` превращается в `backgroundColor`, `font-family` — в `fontFamily`, а `border-radius` — в `borderRadius`.

Объект `letterStyle` и его свойства по большей части представляют собой прямой JavaScript-перевод правила стиля `.letter`, которое мы рассмотрели чуть раньше. Теперь осталось только присвоить этот объект элементу, который мы хотим стилизовать.

Стилизация контента

Теперь, когда у нас есть объект, содержащий стили, которые мы хотим применить, можно сказать, что самое трудное позади. Найдите элемент, к которому вы хотите применить стиль, и установите атрибут `style` для создания ссылки на этот объект. В нашем случае это элемент `div`, возвращаемый функцией `render` компонента `Letter`.

Посмотрите на выделенную цветом строку кода, чтобы понять, как это делается в нашем примере:

```
class Letter extends React.Component {
  render() {
    var letterStyle = {
      padding: 10,
      margin: 10,
      backgroundColor: "#FFDE00",
      color: "#333",
      display: "inline-block",
      fontFamily: "monospace",
```

```
        fontSize: 32,
        textAlign: "center"
      };

      return (
        <div>
          {this.props.children}
        </div>
      );
    }
  }
```

Объект носит имя `letterStyle`, именно его мы и указываем в фигурных скобках, чтобы библиотека React могла оценить выражение. Вот и все. Запустите пример в браузере, чтобы убедиться в том, что все работает и все гласные стилизованы правильно.

Если для дополнительной проверки вы посмотрите стиль, примененный к одной из гласных, используя инструмент разработчика в браузере, то увидите, что стили на самом деле встроенные (рис. 4.4).

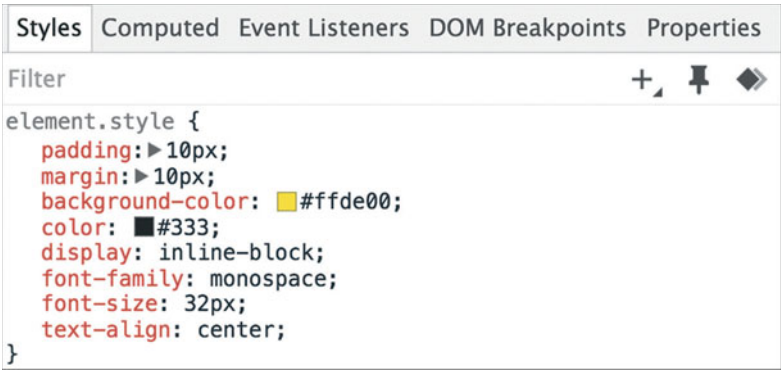


Рис. 4.4. Применение встроенных стилей

К такому подходу может быть трудно адаптироваться, если вы привыкли к тому, что стили находятся внутри соответствующих правил. Но, как говорится, времена меняются.

Настройка фонового цвета

Прежде чем закончить, мы воспользуемся преимуществами того, как библиотека React работает со стилями. Благодаря тому, что стили определены по соседству с JSX, мы можем сделать так, чтобы различные значения стиля легко настраивались предком (потребителем компонента). Давайте посмотрим, как это делается.

В настоящее время все гласные отображаются на желтом фоне. Здорово, если бы цвет фона можно было указать в рамках каждого объявления `Letter`. Чтобы сделать это в методе `ReactDOM.render`, сначала добавьте атрибут `bgcolor` и укажите несколько цветов, как показано в следующих выделенных строках кода:

```
ReactDOM.render(
  <div>
    <Letter bgcolor="#58B3FF">A</Letter>
    <Letter bgcolor="#FF605F">E</Letter>
    <Letter bgcolor="#FFD52E">И</Letter>
    <Letter bgcolor="#49DD8E">O</Letter>
    <Letter bgcolor="#AE99FF">У</Letter>
  </div>,
  destination
);
```

Теперь мы должны использовать это свойство. В объекте `letterStyle` задайте `this.props.bgColor` в качестве значения `backgroundColor`:

```
var letterStyle = {
  padding: 10,
  margin: 10,
  backgroundColor: this.props.bgcolor,
  color: "#333",
  display: "inline-block",
  fontFamily: "monospace",
  fontSize: 32,
```

```
    textAlign: "center"
};
```

Это гарантирует, что значение `backgroundColor` будет выводиться из значения, установленного с помощью атрибута `bgColor` в рамках объявления `Letter`. Если вы просмотрите этот документ в своем браузере, то увидите те же гласные на разноцветном фоне (рис. 4.5).

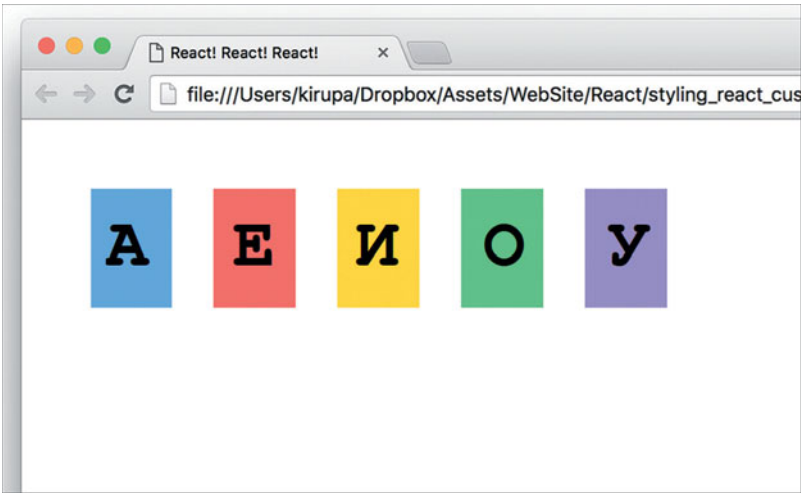


Рис. 4.5. Гласные на разноцветном фоне!

То, что мы сделали, будет очень сложно повторить только с помощью CSS. По мере изучения компонентов, изменение содержимого которых зависит от состояния или пользовательского взаимодействия, мы рассмотрим больше примеров, подчеркивающих достоинства способа стилизации библиотеки React.

Заключение

В процессе более глубокого изучения React вы столкнетесь с еще несколькими случаями, в которых работа этой библиотеки отличается от того, к чему вы привыкли. В этой главе мы видели, как React использует встроенные стили в JavaScript для стилизации контента

вместо правил стилей CSS. Чуть раньше мы рассмотрели JSX и узнали о том, как можно определить весь пользовательский интерфейс в коде JavaScript, используя XML-подобный синтаксис, напоминающий язык HTML.

Если рассмотреть все эти случаи чуть глубже, можно понять, почему способы React расходятся с общепринятыми методами. Создание приложений с очень сложными требованиями к пользовательскому интерфейсу предполагает новый способ решения задач. Методы HTML, CSS и JavaScript, которые были эффективны при работе с веб-страницами и документами, могут быть неприменимы в мире веб-приложений, где одни компоненты повторно используются внутри других.

Учитывая вышесказанное, вам следует тщательно выбирать методы, наиболее подходящие для вашей ситуации. Я склоняюсь к тому, как React решает проблемы разработки пользовательского интерфейса, однако я также стараюсь подчеркнуть альтернативные или общепринятые методы. Возвращаясь к тому, о чем говорилось в этой главе, следует отметить, что применение правил стилей CSS к контенту React абсолютно допустимо при условии, что вы четко представляете себе достоинства и недостатки данного подхода.

> Примечание: Если у вас возникнут какие-либо проблемы, задайте вопрос!
>
> Если у вас есть какие-либо вопросы или код не работает ожидаемым образом, не стесняйтесь задать вопрос! Опубликуйте свой вопрос на форуме **forum.kirupa.com** и получите помощь от самых дружелюбных и знающих людей в Интернете!

Глава 5
Создание сложных компонентов

В главе 3 вы узнали о компонентах и обо всех удивительных вещах, на которые они способны. Вы узнали, что компоненты в React позволяют визуальным элементам вести себя как маленькие многоразовые кирпичики, содержащие весь код HTML, JavaScript и стили, необходимые для запуска. Помимо возможности повторного использования компоненты привносят еще одно важное преимущество. Они предоставляют возможность создавать **композиции**. Вы можете комбинировать компоненты для создания более сложных.

В этой главе мы рассмотрим, что все это значит. Вы изучите и получите следующее:

- Скучные технические детали, которые вам нужно знать.
- Необходимые навыки для определения компонентов среди множества других визуальных элементов.

Отлично, если вы собрались все это читать, но это скучно. Я не обманываю.

От визуализации до компонентов

Примеры, которые мы рассматривали до сих пор, были довольно простыми. Они отлично подходят для демонстрации технических аспектов, но не годятся для подготовки к реальному программированию.

В реальном мире то, что вас попросят реализовать в React, никогда не будет таким простым, как список имен или красочных блоков гласных. Вместо этого вам будет предложено визуализировать какой-нибудь сложный пользовательский интерфейс, содержащий диаграммы, иллюстрации, видео, комментарии и т. п. Затем вам нужно будет оживить все эти статические пиксели. В этой главе вы как раз и попрактикуетесь. Задача — создать простую карточку цветовой палитры (см. рис. 5.1).

Рис. 5.1. Простая карточка цветовой палитры

Карточки цветовой палитры представляют собой небольшие прямоугольные листы, которые помогают сопоставлять цвет с определенным типом краски. Вы можете встретить их в магазинах стройматериалов в отделах подбора краски. У вашего друга-дизайнера наверняка ими набит целый шкаф. Так или иначе, ваша задача — воссоздать одну из этих карточек с помощью React.

Мы могли бы сделать это несколькими способами, но давайте возьмем системный подход, который упростит и поможет понять даже самые сложные пользовательские интерфейсы. Этот подход включает в себя два этапа:

1. Определить основные визуальные элементы.

2. Выяснить, какие будут компоненты.

Оба эти действия звучат довольно сложно, но, познакомившись с ними, вы увидите, как все просто.

Определение основных визуальных элементов

Первый шаг — определить все визуальные элементы, с которыми мы имеем дело. Все визуальные элементы важны. Ни один нельзя пропустить. Самый простой способ определить соответствующие части — начать с очевидных визуальных элементов, а затем заняться менее наглядными. Первое, что вы видите в примере, это сама карточка (см. рис. 5.2).

Рис. 5.2. Карточка

Внутри карточки находятся две отдельных области. Верхняя часть представляет собой прямоугольную область, которая отображает определенный цвет. Нижняя часть — это белая область с шестнадцатеричным значением цвета. Давайте рассмотрим эти два визуальных элемента и разместим их в древовидной структуре, как показано на рис. 5.3.

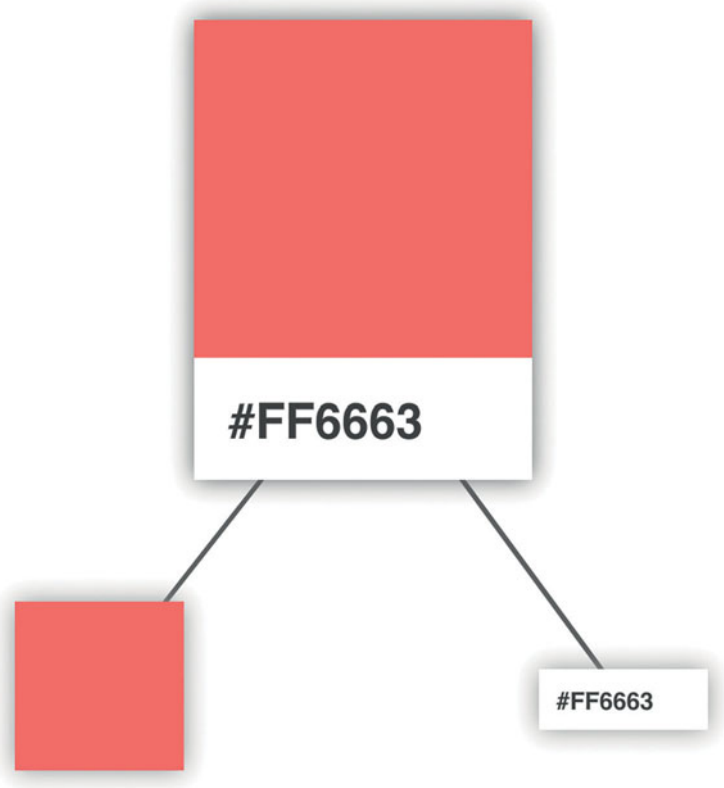


Рис. 5.3. Древовидная структура

Организация визуальных элементов в этой древовидной структуре (т. н. визуальная иерархия) — отличный способ разобраться, как сгруппированы визуальные элементы. Цель этого упражнения — определить важные визуальные элементы и разбивать их на структуру «предок/потомок» до тех пор, пока вы не сможете их разделять далее.

> Примечание: Игнорируйте технические детали реализации.
> Это может быть сложно, но пока не задумывайтесь о деталях реализации. Не думайте о разделении визуальных элементов с учетом кода HTML и CSS. Об этом позже.

Продолжая, мы можем видеть, что окрашенный прямоугольник мы не можем разделить. Это не значит, что мы закончили. Мы также можем отделить метку кода цвета от белой области, на которой метка находится. Теперь визуальная иерархия выглядит так, как показано на рис. 5.4, причем метка и белая область занимают отдельные места в структуре.

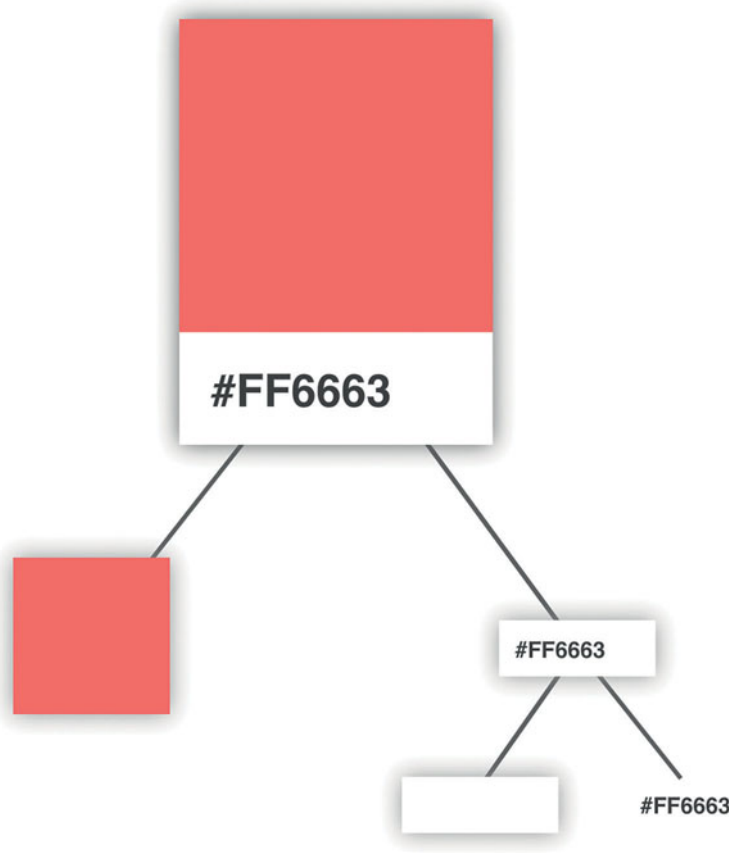


Рис. 5.4. Разделение на метку и белую область, которая ее окружает

На этом этапе нам больше нечего делить. Мы закончили выделять и разделять визуальные элементы, и следующим шагом будет определение компонентов.

Определение компонентов

Теперь становится немного интереснее. Нам нужно выяснить, какой из визуальных элементов, которые мы определили, будет превращен в компонент, а какой нет. Не каждый визуальный элемент должен стать компонентом, но мы также не хотим создавать чрезвычайно сложные компоненты. Нам нужно добиться баланса (см. рис. 5.5).

Рис. 5.5. Правильное количество компонентов

Это целое искусство — определять, какие визуальные элементы входят в состав компонента, а какие нет. *Общее правило состоит в том, что компонент должен делать только что-то одно*. Если вы обнаружите, что ваш потенциальный компонент в итоге делает слишком многое, вы, вероятно, захотите разбить его на несколько компонентов. С другой стороны, если потенциальный компонент имеет небольшой функционал, вы, вероятно, не захотите использовать его по отдельности. Давайте выясним, какие элементы составят компоненты в нашем примере. По визуальной иерархии как карточка, так и цветной квадрат подходят под части компонента. Карточка выполняет роль внешнего контейнера, а цветной квадрат отображает цвет. Метка и окружающая ее белая область остается под вопросом (см. рис. 5.6).

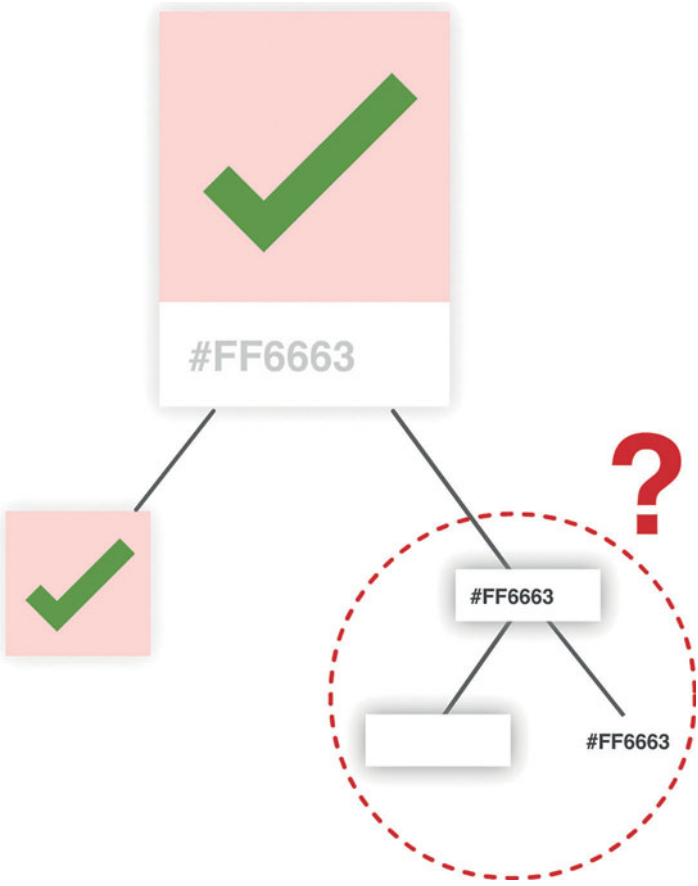


Рис. 5.6. Вопросительный знак вокруг метки и белой области

Важная часть — сама метка. Без нее мы не сможем увидеть шестнадцатеричное значение цвета. Остается белая область. Ее роль незначительна; это лишь пустое пространство, и ее задача легко может быть передана метке. Далее вот что я скажу. К сожалению, белая прямоугольная область не превратится в компонент. На данном этапе мы определили три компонента, и **иерархия компонентов** теперь выглядит так (рис. 5.7):

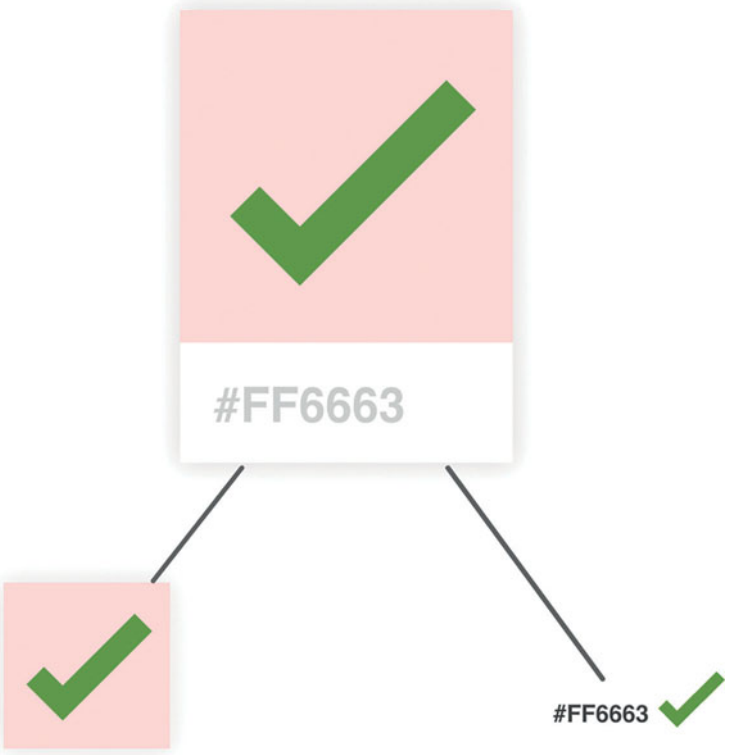


Рис. 5.7. Компоненты структуры

Важно отметить, что иерархия компонентов скорее поможет определить код, чем понять, как будет выглядеть готовый продукт. Как вы могли заметить, результат немного отличается от визуальной иерархии, с которой мы начали. Для визуальных деталей всегда стоит обращаться к исходному контенту (они же ваши визуальные композиции, метки, иллюстрации и другие связанные элементы). Чтобы выяснить, какие компоненты необходимо создать, вы должны использовать иерархию компонентов. Теперь, когда мы определили компоненты и отношения между ними, пришло время вдохнуть в карточку цветовой палитры жизнь.

Создание компонентов

Это самое простое! Пришло время писать код. Для начала нам нужна базовая пустая HTML-страница, которая послужит отправной точкой:

```
<! DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <title>Еще компоненты</title>
  <script src="https://unpkg.com/react@16/umd/react.
      development.js"></script>
  <script src="https://unpkg.com/react-dom@16/umd/react-dom.
      development.js"></script>
  <script src="https://unpkg.com/babel-standalone@6.15.0/
      babel.min.js"></script>

  <style>
    #container {
      padding: 50px;
      background-color: #FFF;
    }
  </style>
</head>

<body>
  <div id="container"></div>

  <script type="text/babel">
    ReactDOM.render(
      <div>

      </div>,
      document.querySelector("#container")
    );
```

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

Обратите внимание, что происходит на этой странице. Не так и много: минимум, необходимый для того, чтобы React визуализировал пустой элемент `div` в элемент **`container`**. Когда вы это сделали, определяем три компонента. Имена, которые мы будем использовать для компонентов, следующие: **`Card`**, **`Label`** и **`Square`**. Далее добавим следующие строки чуть выше функции `ReactDOM.render`:

```
class Square extends React.Component {
  render() {
    return(
      <br/>
    );
  }
}

class Label extends React.Component {
  render() {
    return (
      <br/>
    );
  }
}

class Card extends React.Component {
  render() {
    return (
      <br/>
    );
  }
}
```

Помимо объявления трех компонентов, мы используем функцию `render`, которую необходимо выполнить абсолютно каждому компоненту. Каждая функция `render` возвращает простой элемент `br`; оставив возвращаемое значение для функции `render` пустым, мы получим ошибку. Кроме этого, компоненты сейчас пусты. В следующих разделах мы исправим это, набрав код.

Компонент `Card`

Давайте начнем с вершины иерархии компонентов и сосредоточимся на компоненте `Card`. Этот компонент будет действовать как контейнер, в котором будут находиться компоненты `Square` и `Label`.

Чтобы реализовать это, добавьте выделенный цветом код:

```
class Card extends React.Component {
  render() {
    var cardStyle = {
      height: 200,
      width: 150,
      padding: 0,
      backgroundColor: "#FFF",
      boxShadow: "0px 0px 5px #666"
    };

    return (
      <div style={cardStyle}>

      </div>
    );
  }
}
```

На первый взгляд, нужно внести очень много изменений, но большинство строк форматируют вывод компонента `Card` с помощью объекта `cardStyle`. Остальные изменения незначительны. Мы возвращаем

элемент `div`, и атрибуту `style` этого элемента присваивается объект `cardStyle`. Теперь, чтобы увидеть компонент `Card` в действии, нам нужно отобразить его в модели DOM в составе функции `ReactDOM.render`. Чтобы это произошло, внесите следующие изменения:

```
ReactDOM.render(
  <div>
    <Card/>
  </div>,
  document.querySelector("#container")
);
```

Все, что мы сделали, — это инструктировали функцию `ReactDOM.render` визуализировать вывод компонента `Card`, вызвав его. Если все будет работать правильно, вы увидите результат, идентичный рис. 5.8, после того, как протестируете свое приложение.

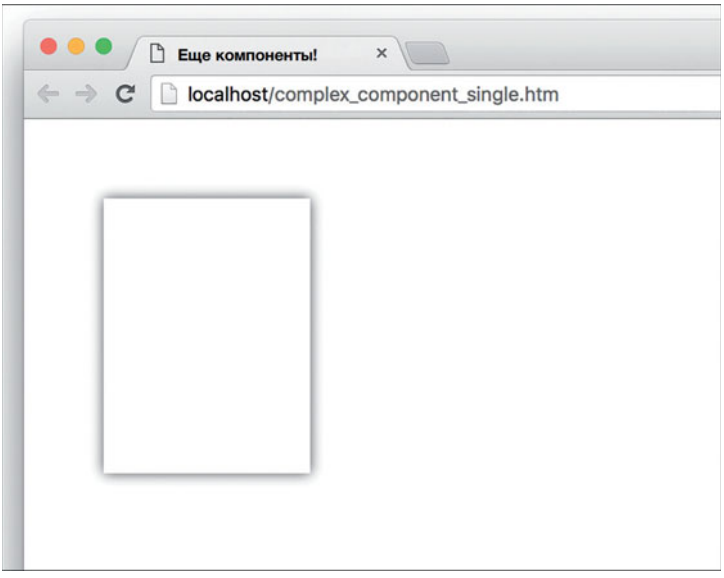


Рис. 5.8. Результат работы программы — контур карточки цветовой палитры

Да, это лишь контур карточки цветовой палитры, но это определенно больше, чем то, с чего вы начали пару минут назад!

Компонент `Square`

Пора спуститься на один уровень вниз в иерархии компонентов и взглянуть на компонент `Square`. Все довольно просто: добавьте следующий выделенный цветом код:

```
class Square extends React.Component {
  render() {
    var squareStyle = {
      height: 150,
      backgroundColor: "#FF6663"
    };

    return (
      <div style={squareStyle}>

      </div>
    );
  }
}
```

Как и в случае с компонентом `Card`, мы возвращаем элемент `div`, атрибуту `style` которого присвоен объект форматирования, который определяет, как выглядит этот компонент. Чтобы увидеть компонент `Square` в действии, нам нужно получить его в модели DOM так же, как и компонент `Card`. Разница на этот раз заключается в том, что мы не будем вызывать компонент `Square` через функцию `ReactDOM.render`. Вместо этого мы вызовем компонент `Square` изнутри компонента `Card`. Чтобы понять, что я имею в виду, вернитесь к функции `render` компонента `Card` и внесите следующие изменения:

```
class Card extends React.Component {
  render() {
    var cardStyle = {
      height: 200,
```

```
        width: 150,
        padding: 0,
        backgroundColor: "#FFF",
        boxShadow: "0px 0px 5px #666"
      };

      return (
        <div style={cardStyle}>
        <Square />
        </div>
      );
    }
  }
```

На этом этапе, если вы запустите приложение, вы увидите окрашенный прямоугольник (см. рис. 5.9).

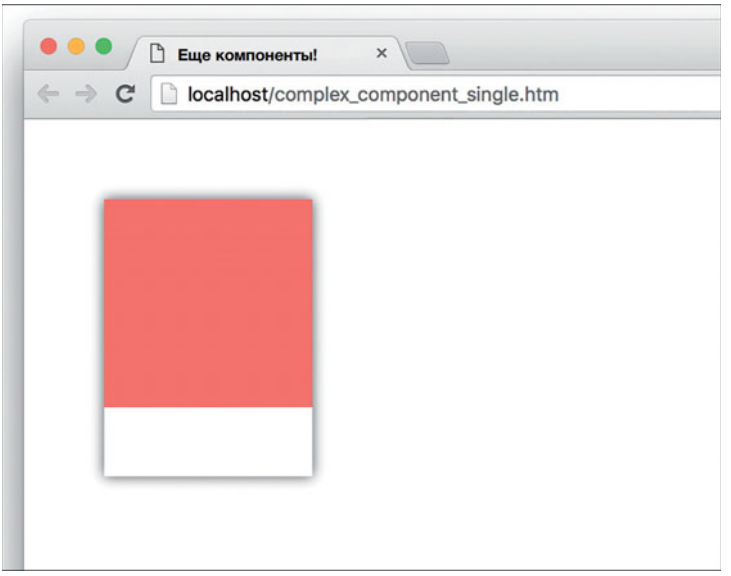


Рис. 5.9. Появилась область, окрашенная в красный цвет

Самое интересное, что мы вызываем компонент `Square` изнутри компонента `Card`! Это пример **компоновки компонентов**,

в котором один компонент связан с выводом другого компонента. В конце вы видите результат совместной работы этих двух компонентов, сговорившихся друг с другом. Разве сговор не прекрасен… хотя бы в этом контексте?

Компонент `Label`

Последний оставшийся компонент — `Label`. Добавьте следующий выделенный цветом код:

```
class Label extends React.Component {
  render() {
    var labelStyle = {
      fontFamily: "sans-serif",
      fontWeight: "bold",
      padding: 13,
      margin: 0
    };

    return (
      <p style={labelStyle}>#FF6663</p>
    );
  }
}
```

Действуем по привычному шаблону. У нас есть объект форматирования, который мы присваиваем возвращаемому элементу. Мы возвращаем элемент `p`, содержимое которого — строка `#FF6663`. Чтобы он в итоге попал в DOM, нам нужно вызвать компонент `Label` через компонент `Card`. Добавьте следующий выделенный цветом код:

```
class Card extends React.Component {
  render() {
    var cardStyle = {
      height: 200,
```

```
        width: 150,
        padding: 0,
        backgroundColor: "#FFF",
        boxShadow: "0px 0px 5px #666"
      };

      return (
        <div style={cardStyle}>
        <Square />
        <Label />
        </div>
      );
    }
  }
```

Обратите внимание, что компонент `Label` располагается сразу под компонентом `Square`, который мы добавили ранее в функцию `return` компонента `Card`. Если вы сейчас запустите приложение в браузере, вы должны увидеть что-то похожее на рис. 5.10.

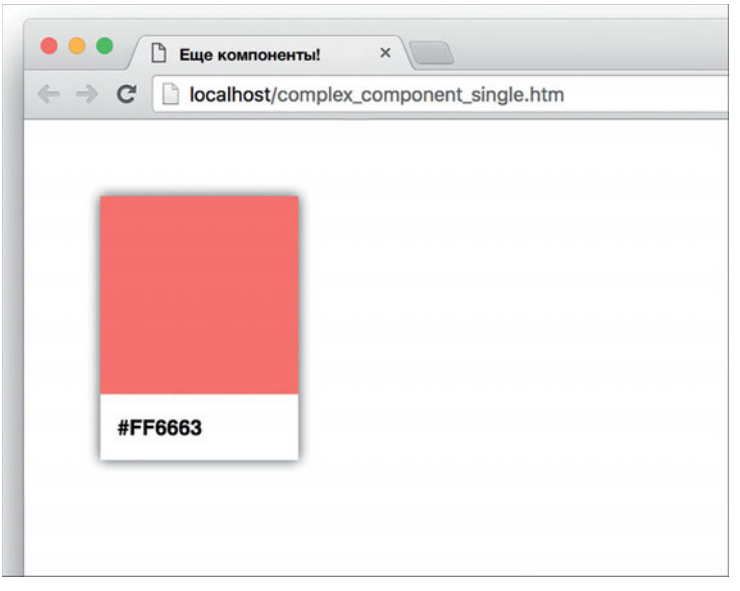


Рис. 5.10. Появилась метка

Ура! Карточка цветовой палитры готова и визуализирована благодаря усилиям компонентов `Card`, `Square` и `Label`. Но это не значит, что мы закончили. Есть еще кое-что.

Передача свойств

В текущем примере мы жестко закодировали значение цвета, используемое компонентами `Square` и `Label`. Это странно и неудобно, но решение проблемы простое. Оно заключается в указании имени свойства и доступ к нему с помощью функции `this.props`. Вы уже применяли это способ раньше; разница только в том, сколько раз нужно это сделать.

Невозможно *правильно* указать свойство на компоненте-предке, чтобы все потомки автоматически получили доступ к этому свойству. Существует много *неправильных* способов решения задачи, например, определение глобальных объектов и непосредственное присвоение значения свойству компонента. Однако мы не будем пользоваться такими неправильными решениями.

Верный способ передачи значения свойства дочернему компоненту состоит в том, чтобы каждый промежуточный компонент-предок передавал свойство. Взгляните на выделенные цветом строки кода. Мы избавляемся от жестко закодированного значения цвета и вместо этого определяем цвет нашей карточки с помощью свойства `color`:

```
class Square extends React.Component {
  render() {
    var squareStyle = {
      height: 150,
      backgroundColor: this.props.color
    };

    return (
      <div style={squareStyle}>

      </div>
```

```
      );
    }
  }

  class Label extends React.Component {
    render() {
      var labelStyle = {
        fontFamily: "sans-serif",
        fontWeight: "bold",
        padding: 13,
        margin: 0
      };

      return (
        <p style={labelStyle}>{this.props.color}</p>
      );
    }
  }

  class Card extends React.Component {
    render() {
      var cardStyle = {
        height: 200,
        width: 150,
        padding: 0,
        backgroundColor: "#FFF",
        boxShadow: "0px 0px 5px #666"
      };

      return (
        <div style={cardStyle}>
          <Square color={this.props.color} />
          <Label color={this.props.color} />
        </div>
      );
```

```
    }
  }

  ReactDOM.render(
    <div>
      <Card color="#FF6663" />
    </div>,
    document.querySelector("#container")
  );
```

После внесения этих изменений вы можете указать шестнадцатеричное значение любого цвета, который хотите использовать, в составе вызова компонента `Card`:

```
  ReactDOM.render(
    <div>
      <Card color="#FFA737"/>
    </div>,
    document.querySelector("#container")
  );
```

Полученная карточка цветовой палитры имеет цвет, указанный вами (рис. 5.11).

Рис. 5.11. Цвет в шестнадцатеричной системе

Теперь давайте вернемся к изменениям, которые мы внесли. Несмотря на то, что свойство `color` нужно только компонентам `Square` и `Label`, компонент-предок `Card` отвечает за передачу свойства им. В случае более глубоко вложенных компонентов используется больше промежуточных компонентов, которые будут отвечать за передачу свойств. Ситуация усложняется. Если есть несколько свойств, которые требуется передать компонентам на нескольких уровнях, объем кода значительно увеличивается. Есть способы упростить код, и мы рассмотрим этот подход более подробно в следующей главе.

Почему так важна компоновка компонентов

Когда мы говорим о React, то часто забываем, что в итоге создаем, — а это простой и скучный код на языках HTML, CSS и JavaScript.

Сгенерированный HTML-код для нашей карточки цветовой палитры выглядит следующим образом:

```
<div id="container">
  <div>
    <div style="height: 200px;
                width: 150px;
                padding: 0px;
                background-color: rgb(255, 255, 255);
                box-shadow: rgb(102, 102, 102) 0px 0px 5px;">
    <div style="height: 150px;
                background-color: rgb(255, 102, 99);">
    </div>
    <p style="font-family: sans-serif;
                font-weight: bold;
                padding: 13px;
                margin: 0px;">
    #FF6663</p>
    </div>
  </div>
</div>
```

Эта разметка совершенно неочевидна. Она не учитывает, какие компоненты за что отвечают. Не определяет компоновку компонентов и не учитывает, что нам пришлось переносить свойство `color` от предка к потомку. Тут возникает важный вывод.

Если обобщить результат работы компонентов, то все, что они делают, — это возвращают определенный HTML-код. Функция `render` каждого компонента возвращает некоторый HTML-код для функции `render` другого компонента. Весь этот HTML-код накапливается до тех пор, пока он не будет передан (очень эффективно) модели DOM. Простота достигается за счет многократного использования отдельных компонентов и отлаженной композиции компонентов. Каждый фрагмент HTML-кода выполняется независимо от других фрагментов HTML-кода, особенно если вы указываете встроенные стили, как рекомендуется делать в React. Это

позволяет легко и без проблем создавать визуальные элементы из других визуальных элементов. Все что угодно! Разве не круто?

Заключение

Как вы поняли, мы постепенно переходим к более продвинутым сценариям, в которых используется React. На самом деле *продвинутые* — это неправильное слово. Правильное слово — *реалистичные*. В этой главе вы начали с изучения того, как анализировать фрагменты пользовательского интерфейса и определять компоненты таким образом, чтобы потом их можно было реализовать. Вы каждый раз будете оказываться в подобной ситуации. В то время как подход, который мы использовали, казался действительно формальным, то, поскольку вы становитесь опытнее в создании различных приложений в React, вы можете избавиться от формальности. Если вы можете быстро определить компоненты и их родственные отношения, не формируя визуальную и компонентную иерархию, это еще один признак того, что вы действительно хорошо освоили React.

Определение компонентов — только одна часть уравнения. Другая часть воплощает эти компоненты в жизнь. Большинство технических материалов, которые вы видели здесь, были лишь незначительным расширением уже известного. Мы рассмотрели один уровень компонентов в предыдущей главе, а здесь выяснили, как работать с несколькими уровнями компонентов. Мы изучили, как передать свойства между одним предком и одним потомком в предыдущей главе, а здесь та же операция затронула уже несколько предков и потомков. Возможно, в будущей главе мы сделаем еще что-то новенькое, например, нарисуем на экране несколько карточек цветовой палитры! Или, может быть, мы сможем указать два свойства вместо одного. Кто знает?

> Примечание: Если у вас возникнут какие-либо проблемы, задайте вопрос!
>
> Если у вас есть какие-либо вопросы или код не работает ожидаемым образом, не стесняйтесь задать вопрос! Опубликуйте свой вопрос на форуме **forum.kirupa.com** и получите помощь от самых дружелюбных и знающих людей в Интернете!

Глава 6
Передача свойств

В работе со свойствами есть и негативная сторона. Мы лишь частично затронули ее в предыдущей главе. Передача свойств из одного компонента в другой — приятная и простая процедура, если вы имеете дело только с одним уровнем вложенности компонентов. Если же вы хотите передать свойство через несколько уровней вложенности, ситуация резко усложняется. Это не сулит ничего хорошего, и в этой главе посмотрим, как мы можем упростить работу со свойствами, передаваемыми через несколько уровней вложенности компонентов.

Обзор проблемы

Предположим, что у вас есть глубоко вложенный компонент, и его иерархия (смоделированная как цветные кружки) выглядит так, как показано на рис. 6.1.

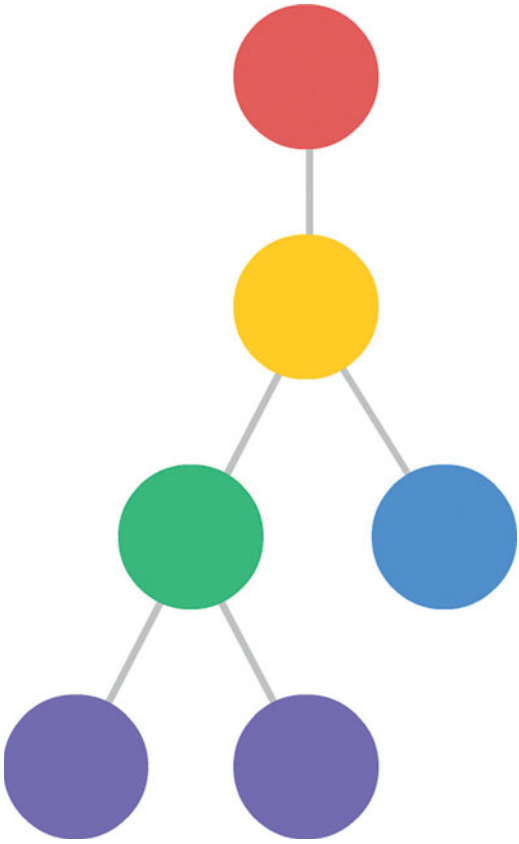


Рис. 6.1. Иерархия компонентов

Вы хотите передать свойство из красного круга фиолетовым, где оно будет использоваться. Вы не сможете так сделать. Это просто понять, взглянув на рис. 6.2.

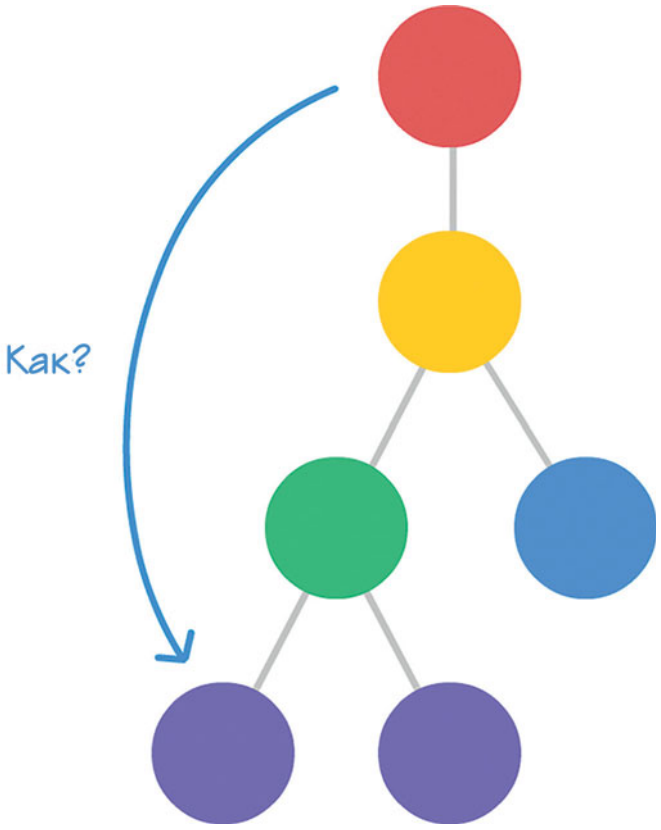


Рис. 6.2. Мы не сможем этого сделать

Вы не можете передать свойство напрямую необходимому компоненту или компонентам. Причина в принципах работы React. *React принудительно применяет цепочку команд, в которой свойства должны передаваться от родительского компонента (предка) к непосредственному дочернему компоненту (потомку)*. Это означает, что вы не можете пропустить дочерний уровень при передаче свойства. Это также означает, что потомки не могут отправить свойство обратно предку. Все связи однонаправлены — от предка к потомку.

В соответствии с этими рекомендациями передача свойств из красного круга в фиолетовый выглядит так, как показано на рис. 6.3.

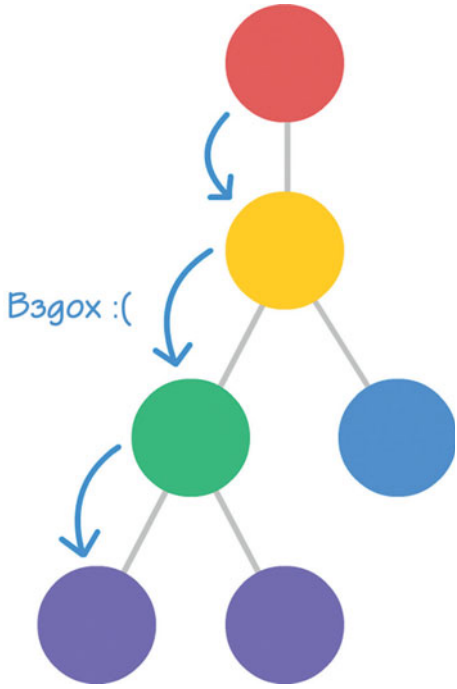


Рис. 6.3. Связь между предком и потомками

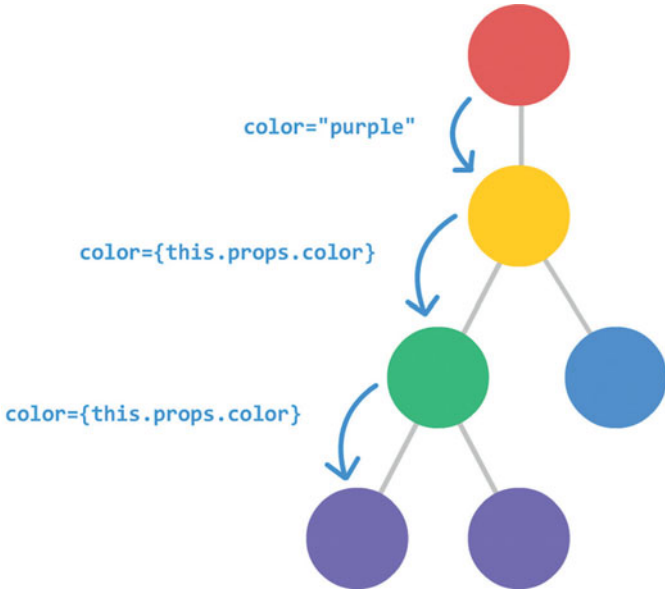


Рис. 6.4. Передача свойства color

Каждый компонент, который лежит на намеченном пути, должен получить свойство от своего предка, а затем передать это свойство своему потомку. Этот процесс повторяется, пока свойство не достигнет своего пункта назначения. Проблема заключается в этапе приема и повторной передаче.

Если бы нам пришлось передать свойство с именем `color` из компонента, представляющего красный круг, в компонент, представляющий фиолетовый круг, его путь к месту назначения выглядел бы примерно так, как показано на рис. 6.4.

Теперь представьте, что у нас есть два свойства, которые нам нужно отправить, как показано на рис. 6.5.

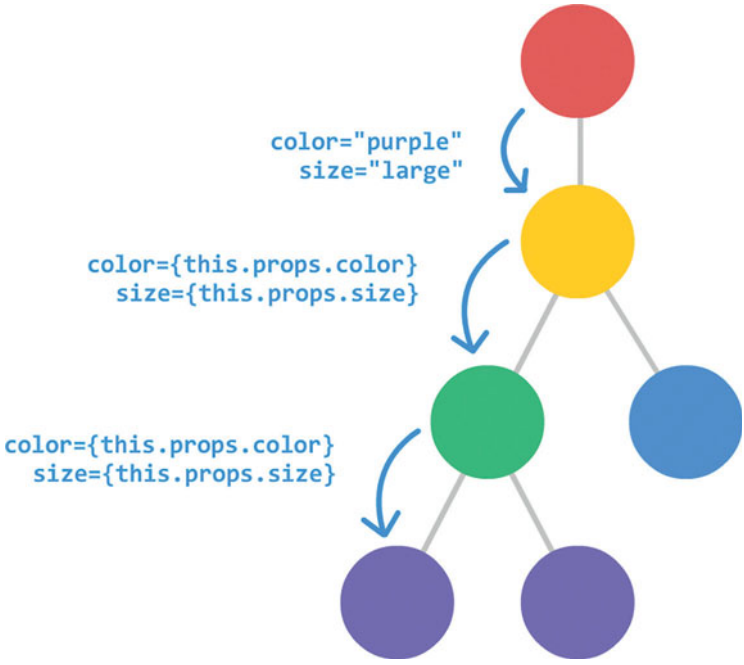


Рис. 6.5. Передача двух свойств

А как быть, если мы хотим отправить три свойства? Или четыре? Как видно из рисунка, этот подход не масштабируемый и не поддерживаемый. Для каждого дополнительного свойства мы должны указывать связь, мы должны добавлять запись с объявлением каждого компонента. Если мы решим переименовать свойства, мы должны

убедиться, что каждый экземпляр этого свойства также переименован. Если мы удалим свойство, нам нужно удалить свойство, которое будет использоваться каждым компонентом, который связан с ним. В целом это те ситуации, которых мы стараемся избегать при написании кода. Как мы можем поступить?

Глубокий анализ проблемы

В предыдущем разделе мы говорили о том, в чем заключается проблема. Прежде чем мы сможем разобраться с решением проблемы, нам нужно отвлечься от диаграмм и рассмотреть подробный пример реального кода. Например:

```
class Display extends React.Component {
  render() {
    return (
      <div>
        <p>{this.props.color}</p>
        <p>{this.props.num}</p>
        <p>{this.props.size}</p>
      </div>
    );
  }
}

class Label extends React.Component {
  render() {
    return (
      <Display color={this.props.color}
               num={this.props.num}
               size={this.props.size}/>
    );
  }
}
```

```
class Shirt extends React.Component {
  render() {
    return (
      <div>
        <Label color={this.props.color}
               num={this.props.num}
               size={this.props.size}/>
      </div>
    );
  }
}

ReactDOM.render(
  <div>
    <Shirt color="steelblue" num="3.14" size="medium" />
  </div>,
  document.querySelector("#container")
);
```

Давайте немного притормозим, чтобы разобраться в происходящем. Затем мы сможем выполнить этот пример вместе.

У нас есть компонент `Shirt`, который зависим от вывода компонента `Label`, который, в свою очередь, зависим от вывода компонента `Display`. (Попробуйте сказать это предложение пять раз очень быстро!) На рис. 6.6 показана иерархия компонентов.

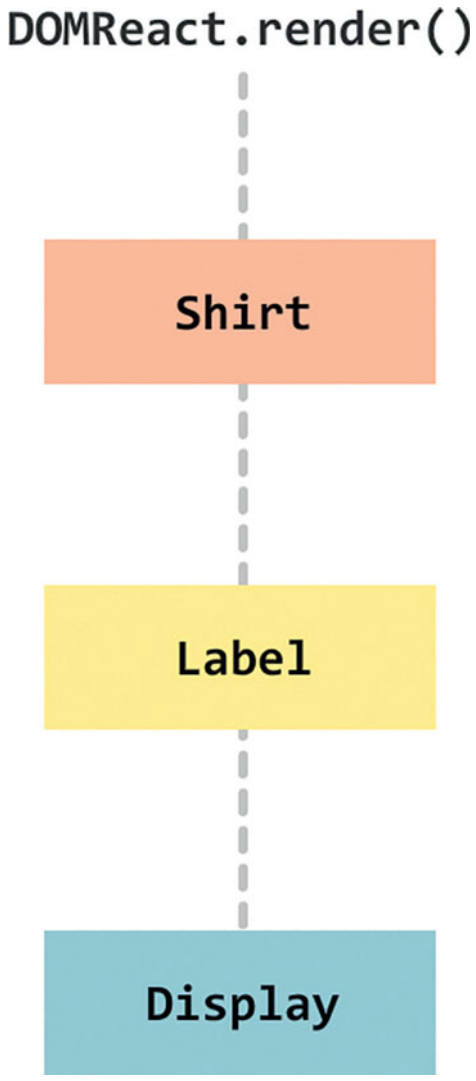


Рис. 6.6. Иерархия компонентов

Когда вы запустите этот код, в выводе не будет ничего особенного. Это всего лишь три строки текста, как показано на рис. 6.7:

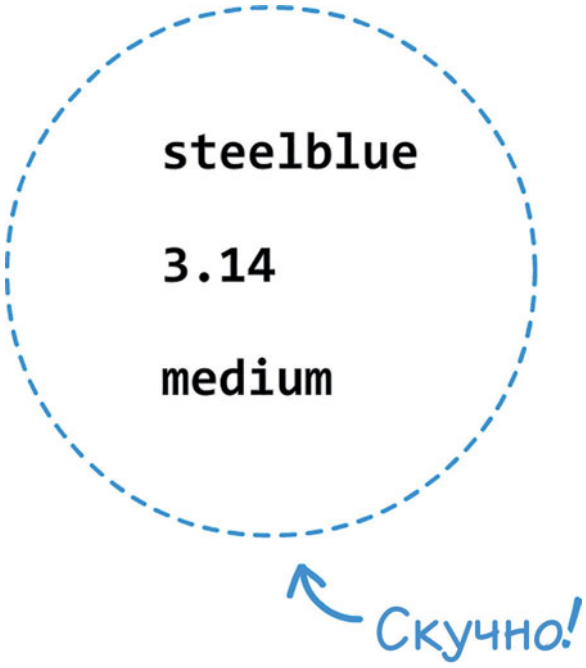


Рис. 6.7. Результат запуска кода

Интересно то, как текст попадает в вывод. Каждая из трех строк текста, которые вы видите, — это свойства, которые мы указали в самом начале кода функции `ReactDOM.render`:

```
<Shirt color="steelblue" num="3.14" size="medium" />
```

Свойства `color`, `num` и `size` (и их значения) путешествуют до компонента `Display`, заставляя ревновать даже самых опытных путешественников в мире. Давайте проследим за этими свойствами с момента их создания, до момента, когда они будут использоваться. (Я понимаю, многое из этого будет повтором показанного раньше. Если вам становится скучно, переходите к следующему разделу.)

Жизнь наших свойств начинается внутри функции `ReactDOM.render`, когда компонент `Shirt` вызывается с указанными свойствами `color`, `num` и `size`:

```
ReactDOM.render(
  <div>
    <Shirt color="steelblue" num="3.14" size="medium" />
  </div>,
  document.querySelector("#container")
);
```

Мы не только определяем свойства, но и инициализируем их значениями, которые они будут нести.

В компоненте `Shirt` эти свойства хранятся внутри объекта `props`. Чтобы передать эти свойства, нам нужно явно получить доступ к этим свойствам из объекта `props` и перечислить их в составе вызова компонента. Ниже приведен пример, как компонент `Shirt` вызывает компонент `Label`:

```
class Shirt extends React.Component {
  render() {
    return (
      <div>
        <Label color={this.props.color}
               num={this.props.num}
               size={this.props.size} />
      </div>
    );
  }
}
```

Обратите внимание, что свойства `color`, `num` и `size` вновь перечислены. Единственное отличие от вызова функции `ReactDOM.render` состоит в том, что значения для каждого свойства берутся из их соответствующей записи в объекте `props`, а не указываются вручную.

Когда компонент `Label` активируется, он имеет свой объект `props`, правильно заполненный сохраненными значениями свойств `color`, `num` и `size`. Вы, вероятно, заметите, что здесь формируется шаблон.

Компонент `Label`, продолжая традицию, повторяет те же шаги и вызывает компонент `Display`:

```
class Label extends React.Component {
  render() {
    return (
      <Display color={this.props.color}
               num={this.props.num}
               size={this.props.size} />
    );
  }
}
```

Обратите внимание, что вызов компонента `Display` содержит тот же список свойств и их значений, взятых из объекта `props` компонента `Label`. Единственная хорошая новость заключается в том, что на этом почти все. Компонент `Display` отображает свойства, поскольку они были заполнены внутри объекта `props`:

```
class Display extends React.Component {
  render() {
    return (
      <div>
        <p>{this.props.color}</p>
        <p>{this.props.num}</p>
        <p>{this.props.size}</p>
      </div>
    );
  }
}
```

Вот! Все, что мы хотели сделать, это посмотреть, как компонент `Display` отображает значения переменных `color`, `num` и `size`. Единственная сложность состояла в том, что значения, которые мы хотели вывести, первоначально были определены в функции

`ReactDOM.render`. Досадно. При этом каждый компонент по пути к месту назначения должен получить доступ и переопределить каждое свойство. Это ужасно. Мы можем выбрать иной, лучший путь, который вы увидите через пару мгновений.

Оператор расширения

Решение всех наших проблем заключается в новом для JavaScript приеме, известном как **оператор расширения**. Принцип работы оператора расширения сложновато объяснить без примера, поэтому давайте рассмотрим код, а затем разберем определение.

Взгляните на следующий фрагмент кода:

```
var items = ["1", "2", "3"];

function printStuff(a, b, c) {
  console.log("Вывести: " + a + " " + b + " " + c);
}
```

У нас есть массив с именем `items`, который содержит три значения. У нас также есть функция `printStuff`, которая принимает три аргумента. Мы хотим указать три значения из массива `items` в качестве аргументов функции `printStuff`. Звучит довольно просто, не так ли?

Вот один из наиболее распространенных способов сделать это:

```
printStuff(items[0], items[1], items[2]);
```

Мы получаем доступ к каждому элементу массива по отдельности и передаем его функции `printStuff`.

А с оператором расширения нам доступен более простой способ. Нам не нужно указывать каждый элемент массива по отдельности; мы можем поступить так:

```
printStuff(…items);
```

Оператор расширения — это символы ... перед массивом элементов. Использование кода ...items идентично вызову элементов по отдельности, items [0], items [1] и items [2], как мы это делали ранее. При запуске функция printStuff выведет в оболочке командной строки числа 1, 2 и 3. Довольно круто, не так ли?

Теперь, когда вы опробовали оператор расширения, пришло время его определить. *Оператор расширения позволяет развернуть массив на отдельные элементы.* Оператор расширения делает еще кое-что, но пока это неважно. Мы будем использовать только эту описанную функцию оператора расширения для решения проблемы с передачей свойств.

Лучший способ передачи свойств

Вы видели пример использования оператора расширения, чтобы избавиться от необходимости перечислять каждый отдельный элемент массива при передаче его функции:

```
var items = ["1", "2", "3"];

function printStuff(a, b, c) {
  console.log("Вывести: " + a + " " + b + " " + c);
}

// с оператором расширения
printStuff(…items);

// без оператора расширения
printStuff(items[0], items[1], items[2]);
```

Ситуация, с которой мы сталкиваемся при передаче свойств компонентам, очень похожа на нашу проблему доступа к каждому элементу массива по отдельности. Позвольте мне уточнить.

Внутри компонента объект props выглядит следующим образом:

```
var props = {
  color: "steelblue",
  num: "3.14",
  size: "medium"
};
```

В рамках передачи этих значений свойства дочернему компоненту мы вручную обращаемся к каждому элементу объекта `props`:

```
<Display color={this.props.color}
         num={this.props.num}
         size={this.props.size}/>
```

Было бы здорово, если бы был способ развернуть объект и передать пары свойство/значение так же, как мы смогли развернуть массив с помощью оператора расширения!

Как оказалось, способ есть. На это способен сам оператор с расширением. Объясню позже, но на данный момент это означает, что мы можем вызвать компонент `Display`, используя код `…this.props`:

```
<Display {…this.props} />
```

Поведение во время выполнения при использовании кода `…this.props` такое же, что и при указании свойств `color`, `num` и `size` вручную. Это означает, что предыдущий пример можно упростить следующим образом (обратите внимание на выделенные строки):

```
class Display extends React.Component {
  render() {
    return (
      <div>
        <p>{this.props.color}</p>
        <p>{this.props.num}</p>
        <p>{this.props.size}</p>
      </div>
    );
```

```
    }
  }

  class Label extends React.Component {
    render() {
      return (
        <Display {…this.props} />
      );
    }
  }

  class Shirt extends React.Component {
    render() {
      return (
        <div>
          <Label {…this.props} />
        </div>
      );
    }
  }
```

Если мы запустим этот код, конечный результат не изменится по сравнению с прошлыми примерами. Главное отличие — мы больше не передаем расширенные формы каждого свойства в составе вызова каждого компонента. Это решает все проблемы, которые мы собирались решить.

Используя оператор расширения (если вы когда-нибудь решите добавить свойства, переименовать их или удалить, либо же выполнить какие-либо другие связанные со свойствами действия), вам не нужно будет вносить миллиард различных изменений. Вы вносите одно изменение в позиции, где определяете свое свойство. И вносите другое изменение в позиции, где применяете свойство. И все. Все промежуточные компоненты, которые переносят свойства, остаются нетронутыми, потому что выражение `{…this.props}` не содержит информации о том, что происходит внутри него.

> Это лучший способ передачи свойств?
>
> Использовать оператор расширения для передачи свойств удобно, и это заметно лучше по сравнению с явным определением каждого свойства каждого компонента, как мы делали изначально. Но дело в том, что даже подход с оператором расширения не идеален. Если все, что вы хотите сделать, это передать свойство определенному компоненту, если каждый промежуточный компонент играет пассивную роль в его передаче, оператор не необходим. Хуже того, у него есть все шансы негативно повлиять на производительность. Любое изменение свойства приведет к обновлению каждого компонента на пути его следования. Это нехорошо! Позже мы рассмотрим способы решения этой проблемы — переноса свойств без каких-либо побочных эффектов.

Заключение

Разработанный комитетом ES6/ES2015 оператор расширения предназначен для работы только с массивами или массивоподобными структурами (что-то, что имеет свойство `Symbol.iterator`). Тот факт, что он работает с объектными литералами, такими как объект `props`, является результатом того, что это расширение стандарта React. В настоящее время браузеры не поддерживают использование объекта расширения в объектных литералах. Наш пример функционирует благодаря транспилятору Babel. Помимо превращения всего JSX-кода в код, поддерживаемый браузером, Babel превращает передовые и экспериментальные функции в браузерах в удобные и рабочие инструменты. Вот почему мы можем избавиться от оператора расширения в объектном литерале, и именно поэтому возможно изящно решить проблему переноса свойств на нескольких уровнях вложенности компонентов.

Так что-нибудь из этого имеет значение? Действительно ли важно знать о нюансах оператора расширения и о том, как он работает в определенных ситуациях и не работает в других? По большей части нет. Важный момент реализации заключается в том, что вы можете

использовать оператор расширения для передачи объекта `props` с одного компонента на другой. Другой важный момент реализации заключается в том, что в будущем мы рассмотрим другие пути, чтобы сделать передачу свойств одинаково простой, без каких-либо снижений производительности.

> Примечание: Если у вас возникнут какие-либо проблемы, задайте вопрос!
>
> Если у вас есть какие-либо вопросы или код не работает ожидаемым образом, не стесняйтесь задать вопрос! Опубликуйте свой вопрос на форуме **forum.kirupa.com** и получите помощь от самых дружелюбных и знающих людей в Интернете!

Глава 7
Встречайте JSX… Вновь!

Как вы уже заметили, мы использовали много JSX-кода. Но реально не разобрались, что такое **синтаксический сахар JSX**. Как он работает? Почему бы нам не назвать его HTML-кодом? Какие тайны он скрывает? В этой главе я отвечу на все эти вопросы и на многие другие! Мы сделаем приличный шаг назад (и пару шажочков вперед), чтобы разузнать все необходимое о JSX, чтобы быть уверенным в своих познаниях.

Что происходит с JSX?

Один из самых важных вопросов, который еще не освещен, — это выяснить, что происходит с JSX-кодом после того, как мы его написали. Как получается HTML-код, который вы видите в браузере? Взгляните на следующий пример, где определяется компонент с именем `Card`:

```
class Card extends React.Component {
  render() {
    var cardStyle = {
      height: 200,
      width: 150,
      padding: 0,
      backgroundColor: "#FFF",
      boxShadow: "0px 0px 5px #666"
    };
```

```
    return (
      <div style={cardStyle}>
        <Square color={this.props.color} />
        <Label color={this.props.color} />
      </div>
    );
  }
}
```

Можно быстро определить код JSX. Это четыре строки, показанные ниже:

```
<div style={cardStyle}>
  <Square color={this.props.color} />
  <Label color={this.props.color} />
</div>
```

Имейте в виду, что браузеры не знают, что делать с JSX. Вероятно, они сочтут вас сошедшими с ума, если вы предложите им обработать JSX. Вот почему мы полагаемся на такие инструменты, как Babel, чтобы превратить JSX-код в то, что понимают браузеры: JavaScript.

Это означает, что JSX-код, который мы пишем, предназначен только для человека. Когда JSX-код достигает браузера, он результирует превращением в обычный JavaScript-код:

```
return React.createElement(
  "div",
  {style: cardStyle},
  React.createElement(Square, {color: this.props.color}),
  React.createElement(Label, {color: this.props.color})
);
```

Все эти аккуратно вложенные HTML-подобные элементы, их атрибуты и потомки превращаются в серию вызовов функции

`createElement` с дефолтными значениями инициализации. Вот как выглядит весь код компонента `Card`, превращенный в JavaScript:

```
class Card extends React.Component {
  render() {
    var cardStyle = {
      height: 200,
      width: 150,
      padding: 0,
      backgroundColor: "#FFF",
      boxShadow: "0px 0px 5px #666"
    };

    return React.createElement(
      "div",
      {style: cardStyle},
      React.createElement(Square, {color: this.props.color}),
      React.createElement(Label, {color: this.props.color})
    );
  }
}
```

Обратите внимание, что нигде нет JSX-кода! Все эти изменения между тем, что мы написали, и тем, что получает браузер, — часть операции, о которой мы говорили в первой главе. Эта трансформация происходит полностью за кадром, благодаря инструменту Babel, который мы использовали для полного преобразования JSX-JS в браузере. В конечном итоге мы рассмотрим применение Babel в составе более сложной среды сборки, в которой мы создадим преобразованный JS-файл, но об этом позже. Итак, у вас есть ответ на то, что именно происходит с JSX-кодом: он превращается в приятный JavaScript-код.

Причуды JSX, которые надо запомнить

Работая с JSX, вы, вероятно, столкнулись с некоторыми необычными правилами и исключениями из того, что мы можем и чего не можем сделать. В этом разделе взглянем на эти и некоторые другие причуды!

Обработка выражений

Код JSX обрабатывается как JavaScript. Как вы уже видели не раз, это означает, что вы не ограничены работой со статическим контентом, например:

```
class Stuff extends React.Component {
  render() {
    return (
      <h1>Скучный статический контент!</h1>
    );
  }
};
```

Возвращаемые значения могут быть сгенерированы динамически. Все, что вам нужно сделать, это заключить выражение в фигурные скобки:

```
class Stuff extends React.Component {
  render() {
    return (
      <h1>Скучный {Math.random() * 100} контент!</h1>
    );
  }
}
```

Обратите внимание, как мы используем вызов функции `Math.random()` для генерации случайного числа. Он обрабатывается вместе со статическим текстом по бокам, но из-за фигурных скобок вы

увидите примерно следующее: `Скучный 28.6388820148227 контент!`

Эти фигурные скобки позволяют приложению сначала обработать выражение, а затем вернуть результат обработки. Без скобок вы увидите, что выражение возвращается как текст: `Скучный Math.random() * 100 контент!`

Это, вероятно, не то, что вы хотите получить.

Возвращение нескольких элементов

Во многих наших примерах мы возвращали один элемент верхнего уровня (`div`), который включал много других элементов. Здесь нет ограничений: фактически вы можете вернуть несколько элементов и сделать это двумя способами.

Один из способов — использовать массивоподобный синтаксис:

```
class Stuff extends React.Component {
  render() {
    return (
      [
        <p>Я</p>,
        <p>возвращаю список</p>,
        <p>элементов!</p>
      ]
    );
  }
}
```

Здесь мы возвращаем три элемента `p`. У них нет единого предка.

Теперь, когда вы возвращаете несколько элементов, вы можете иметь дело или не иметь с одним элементом, в зависимости от версии React, которую используете. Вам нужно указать атрибут `key` и уникальное значение для каждого элемента:

```
class Stuff extends React.Component {
  render() {
    return (
      [
        <p key="1">Я</p>,
        <p key="2">возвращаю список</p>,
        <p key="3">элементов!</p>
      ]
    );
  }
}
```

Так React проще понять, с каким элементом он имеет дело, и вносить какие-либо изменения в него. Как узнать, нужно ли добавлять атрибут `key`? React подскажет вам. Вы увидите сообщение, похожее на следующее, выведенное в оболочке командной строки: *Warning: Each child in an array or iterator should have a unique "key" prop — Внимание: Каждый потомок в массиве или итераторе должен иметь уникальное свойство "key"*.

Существует также другой (и, возможно, лучший) способ возврата нескольких элементов. Он связан с так называемыми фрагментами. Его применяют следующим образом:

```
class Stuff extends React.Component {
  render() {
    return (
      <React.Fragment>
      <p>Я</p>
      <p>возвращаю список</p>
      <p>элементов!</p>
      </React.Fragment>
    );
  }
}
```

Вы оборачиваете список элементов, который хотите вернуть, в магический компонент `React.Fragment`. Обратите внимание на несколько интересных моментов:

1. Этот компонент фактически не генерирует элемент DOM. Вы лишь указываете его в JSX-коде, который преобразуется в HTML-код, поддерживаемый браузером.

2. Вы не указываете конкретно, что возвращаете элементы массива, поэтому вам не нужны запятые или другие символы, разделяющие элементы.

3. Нет необходимости указывать уникальный атрибут и значение `key`; все это будет сделано за вас.

Прежде чем закончить этот раздел, запомните, что вы можете использовать более лаконичный синтаксис вместо полного указания компонента `React.Fragment`. Вы можете использовать только символы <> и </>:

```
class Stuff extends React.Component {
  render() {
    return (
      <>
        <p>Я</p>
        <p>возвращаю список</p>
        <p>элементов!</p>
      </>
    );
  }
}
```

Это словно футуристический код, поэтому, если вам понравилось использовать фрагменты для возврата нескольких значений, не стесняйтесь.

Нельзя использовать встроенные стили CSS

Как вы видели в главе 4, атрибут `style` в JSX-коде ведет себя иначе, чем его брат-близнец в HTML-коде. В HTML-коде вы можете напрямую указать свойства CSS в качестве значений атрибута `style`:

```
<div style="font-family: Arial; font-size:24px">
  <p>Ой!</p>
</div>
```

В JSX-коде атрибут `style` не может содержать каскадные таблицы стилей. Вместо этого он должен ссылаться на объект, содержащий информацию о стиле:

```
class Letter extends React.Component {
  render() {
    var letterStyle = {
      padding: 10,
      margin: 10,
      backgroundColor: this.props.bgcolor,
      color: "#333",
      display: "inline-block",
      fontFamily: "monospace",
      fontSize: "32",
      textAlign: "center"
    };

    return (
      <div style={letterStyle}>
        {this.props.children}
      </div>
    );
  }
}
```

Обратите внимание, что используется объект с именем `letterStyle`, который содержит имена свойств CSS (в горбатом регистре в духе JavaScript) и их значения. Этот объект мы указываем в качестве атрибута `style`.

Комментарии

Так же, как и в языках HTML, CSS и JavaScript, не будет лишним снабдить комментариями и JSX-код. Написание комментариев в JSX аналогично тому, как вы это делаете в JavaScript, за одним исключением. Если вы указываете комментарий как дочерний элемент элемента, вам нужно заключить комментарий в скобки `{}`, чтобы убедиться, что он анализируется как выражение:

```
ReactDOM.render(
  <div className="slideIn">
    <p className="emphasis">Гоооол!</p>
    {/* Дочерний комментарий */}
    <Label/>
  </div>,
  document.querySelector("#container")
);
```

Комментарий в этом случае — потомок элемента `div`. Как вариант, можно указать однострочный или многострочный комментарий целиком внутри тега элемента, не используя фигурные скобки `{}`:

```
ReactDOM.render(
  <div className="slideIn">
    <p className="emphasis">Гоооол!</p>
    <Label
      /* Этот комментарий
         состоит из
         нескольких строк */
    className="colorCard" // конец строки
```

```
      />
    </div>,
    document.querySelector("#container")
);
```

В этом листинге приведен пример как многострочного комментария, так и однострочного — в конце строки. Теперь, когда вы все это знаете, вы просто обязаны комментировать свой JSX-код.

Регистр, HTML-элементы и компоненты

Регистр очень важен. Чтобы указать имена HTML-элементов, убедитесь, что теги HTML-элементов набраны строчными буквами:

```
ReactDOM.render(
  <div>
    <section>
      <p>Какой-то контент!</p>
    </section>
  </div>,
  document.querySelector("#container")
);
```

Если вы хотите указать имена компонентов, они должны быть написаны с прописной буквы:

```
ReactDOM.render(
  <div>
    <MyCustomComponent/>
  </div>,
  document.querySelector("#container")
);
```

Если вы ошиблись с регистром, React не сможет правильно визуализировать контент. О проблеме с регистром, как правило,

вспоминают в последнюю очередь, когда что-то не работает, поэтому помните об этом небольшом совете.

JSX-код может быть где угодно

Во многих ситуациях JSX-код не будет аккуратно размещаться внутри функции `render` или `return`, как в примерах, которые вы видели до сих пор. Взгляните на следующий пример:

```
var swatchComponent = <Swatch color="#2F004F"></Swatch>;
ReactDOM.render(
  <div>
    {swatchComponent}
  </div>,
  document.querySelector("#container")
);
```

Здесь есть переменная `swatchComponent`, которая инициализируется как строка JSX-кода. Когда переменная `swatchComponent` помещается внутрь функции `render`, инициализируется компонент `Swatch`. Все это абсолютно справедливо. В будущем вы будете выполнять больше таких действий, когда узнаете, как создавать и обрабатывать JSX-код с помощью JavaScript.

Заключение

В этой главе мы наконец собрали в одном месте различные обрывки информации о JSX, которые были представлены в предыдущих главах. Самый важный момент, который следует помнить: JSX — это *не* HTML. Он похож на HTML-код и ведет себя во многих случаях аналогично, но в итоге он предназначен для преобразования в JavaScript. Это означает — вы можете делать то, что никогда не могли себе представить, верстая обычный HTML-код. Возможность обрабатывать выражения или программно манипулировать целыми фрагментами

JSX-кода — это только начало. В следующих главах мы рассмотрим это пересечение технологий JavaScript и JSX.

> Примечание: Если у вас возникнут какие-либо проблемы, задайте вопрос!
> Если у вас есть какие-либо вопросы или код не работает ожидаемым образом, не стесняйтесь задать вопрос! Опубликуйте свой вопрос на форуме **forum.kirupa.com** и получите помощь от самых дружелюбных и знающих людей в Интернете!

Глава 8
Работа с состояниями React

До этого момента компоненты, которые мы создали, не имели состояний. У них есть свойства (`props`), которые передаются от их предка, но ничто (обычно) не изменяет их после того, как компоненты начинают работать. Объекты считаются *неизменяемыми* после их установки. Для многих интерактивных сценариев это недопустимо. Нужна возможность изменять аспекты компонентов в результате взаимодействия с пользователем (или некоторые данные, возвращаемые с сервера, или миллиарды других вещей).

Нам нужен другой способ хранения данных в компоненте, выходя за рамки свойств. Нам нужен способ хранения данных, чтобы их можно было изменять. Нам нужно что-то, известное как **состояние**. В этой главе вы узнаете все о состояниях и о том, как вы можете использовать их для создания изменяемых компонентов.

Использование состояний

Если вы знаете, как работать со свойствами, вы полностью знаете, как работать с состояниями… типа того. Есть некоторые отличия, но они слишком мелкие, чтобы вас утомлять ими прямо сейчас. Вместо этого давайте опробуем состояния в действии, используя их в небольшом примере.

Мы создадим простой пример: счетчик молний, как показано на рис. 8.1.

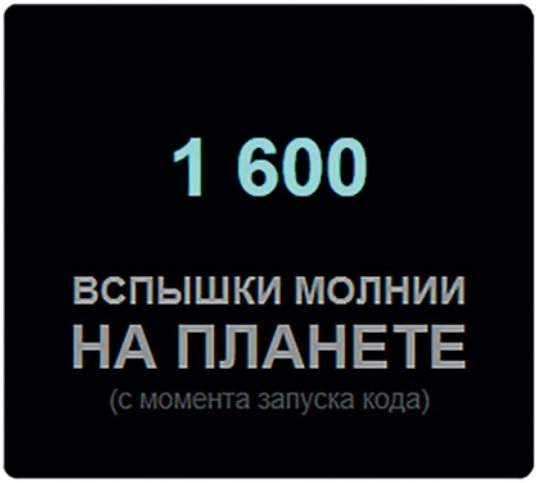


Рис. 8.1. Приложение, которое вы создадите

Этот пример не делает ничего сверхъестественного. По данным журнала `National Geographic`, молния поражает земную поверхность около 100 раз в секунду. У нас есть счетчик, который увеличивает число, которое вы видите, с тем же коэффициентом. Давайте создадим приложение.

Начало работы

Основное внимание в этом примере — посмотреть, как мы можем работать с состояниями. Нет смысла тратить много времени на разработку примера с нуля и всего остального, что мы уже делали много раз. Это не лучшая трата времени.

Вместо того чтобы начинать с нуля, измените существующий HTML-документ или создайте новый со следующим содержимым:

```
<! DOCTYPE html>
<html>

<head>
  <meta charset="utf-8">
```

```
    <title>Состояния</title>
    <script src="https://unpkg.com/react@16/umd/react.
        development.js"></script>
    <script src="https://unpkg.com/react-dom@16/umd/react-dom.
        development.js"></script>
    <script src="https://unpkg.com/babel-standalone@6.15.0/
        babel.min.js"></script>
</head>

<body>
    <div id="container"></div>

    <script type="text/babel">
      class LightningCounter extends React.Component {
        render() {
          return (
            <h1>Привет!</h1>
          );
        }
      }

      class LightningCounterDisplay extends React.Component {
        render() {
          var divStyle = {
            width: 250,
            textAlign: "center",
            backgroundColor: "black",
            padding: 40,
            fontFamily: "sans-serif",
            color: "#999",
            borderRadius: 10
          };

          return (
            <div style={divStyle}>
```

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

    ReactDOM.render(
      <LightningCounterDisplay/>,
      document.querySelector("#container")
    );
  </script>
</body>

</html>
```

Теперь посмотрим, что делает этот код. Во-первых, у нас есть компонент с именем `LightningCounterDisplay`. Основная часть этого компонента — это объект `divStyle`, который содержит информацию о стиле, отвечающую за прикольный округлый фон. Функция `return` возвращает элемент `div`, в который обернут компонент `LightningCounter`.

Компонент `LightningCounter` — это место, где все действия будут выполняться:

```
class LightningCounter extends React.Component {
  render() {
    return (
      <h1>Привет!</h1>
    );
  }
}
```

На данном этапе этот компонент не делает ничего интересного. Он возвращает слово **Привет!**. Все в порядке — мы исправим этот компонент позже.

Последнее, что нужно рассмотреть, это метод `ReactDOM.render`:

```
ReactDOM.render(
  <LightningCounterDisplay/>,
  document.querySelector("#container")
);
```

Он проталкивает компонент `LightningCounterDisplay` в элемент `container` в модели DOM. Это действительно так. В конце концов получаем комбинацию разметки из метода `ReactDOM.render` и компонентов `LightningCounterDisplay` и `LightningCounter`.

Настройка счетчика

Теперь, когда у вас есть представление о том, с чего мы начали, пришло время запланировать дальнейшие шаги. То, как работает счетчик, довольно легко понять. Мы будем использовать функцию `setInterval`, которая вызывает определенный код каждые 1000 миллисекунд (1 секунду). Этот «определенный код» будет увеличивать значение на 100 каждый раз, когда вызывается. Кажется, довольно просто, не так ли?

Чтобы все заработало, мы полагаемся на API-интерфейсы, которые предоставляет компонент React:

1. `componentDidMount`
   Этот метод вызывается сразу *после* того, как компонент визуализируется (рендерится) (или **монтируется** на языке React).

2. `setState`
   Этот метод позволяет обновить значение `state` объекта.

Вы скоро увидите эти API-интерфейсы в действии, а сейчас познакомимся с ними бегло.

Установка начального значения состояния

Нам нужна переменная, которая будет выполнять роль счетчика. Присвоим этой переменной имя `strikes`. Существует куча способов создать такую переменную, но наиболее очевидный из них следующий:

```
var strikes = 0; // :P
```

Мы не хотим этого делать. В нашем примере переменная `strikes` является частью состояния компонента. Нам нужно создать объект `state`, сделать переменную `strikes` его свойством и убедиться, что мы это все сделали до того, как компонент будет создан. Компонент, который мы хотим сделать, носит имя `LightningCounter`. Внесите следующие изменения (выделены цветом):

```
class LightningCounter extends React.Component {
  constructor(props) {
    super(props);

    this.state = {
      strikes: 0
    };
  }

  render() {
    return (
      <h1>Привет!</h1>
    );
  }
}
```

Мы указываем объект `state` внутри конструктора компонента `LightningCounter`. Это действие должно быть выполнено до того, как компонент будет визуализирован. Мы сообщаем React, чтобы он создал объект, содержащий свойство `strikes` (присвоено значение `0`).

Если мы проверим значение объекта `state` после запуска этого кода, результат будет таким:

```
var state = {
  strikes: 0
};
```

Прежде чем мы закончим этот раздел, давайте визуализируем свойство `strikes`. В коде метода `render` внесите следующие изменения (выделены цветом):

```
class LightningCounter extends React.Component {
  constructor(props) {
    super(props);

    this.state = {
      strikes: 0
    };
  }

  render() {
    return (
      <h1>{this.state.strikes}</h1>
    );
  }
}
```

Мы заменили слово `Привет!` на выражение, которое отображает значение, сохраненное свойством `this.state.strikes`. Если вы просмотрите свой пример в браузере, вы увидите значение `0`. Это только начало!

Запуск таймера и установка состояния

Далее, мы получаем значение таймера и увеличиваем значение свойства `strikes`. Как упоминалось ранее, мы будем использовать функцию `setInterval`, чтобы увеличивать значение свойства `strikes` на 100 каждую секунду. Мы собираемся сделать все это сразу после того, как компонент будет визуализирован с помощью встроенного метода `componentDidMount`.

Код запуска таймера выглядит следующим образом:

```
class LightningCounter extends React.Component {
  constructor(props) {
    super(props);

    this.state = {
      strikes: 0
    };
  }

  componentDidMount() {
    setInterval(this.timerTick, 1000);
  }

  render() {
    return (
      <h1>{this.state.strikes}</h1>
    );
  }
}
```

Далее добавим эти выделенные строки в код нашего приложения. Внутри метода `componentDidMount`, который вызывается после того, как компонент рендерится (визуализируется), у нас есть метод `setInterval`, который вызывает функцию `timerTick` каждую секунду (или 1000 миллисекунд).

Мы не определили функцию `timerTick`, поэтому давайте исправим это, внеся следующие изменения (выделены цветом) в код:

```
class LightningCounter extends React.Component {
  constructor(props) {
    super(props);

    this.state = {
      strikes: 0
    };
  }

  timerTick() {
    this.setState({
      strikes: this.state.strikes + 100
    });
  }

  componentDidMount() {
    setInterval(this.timerTick, 1000);
  }

  render() {
    return (
      <h1>{this.state.strikes}</h1>
    );
  }
}
```

Что делает функция `timerTick`, довольно легко понять: она лишь вызывает метод `setState`. Этот метод используется в разных случаях, но в нашем примере он принимает объект в качестве аргумента. Этот объект содержит все свойства, которые вы хотите *свести* в *объект* `state`. В нашем случае мы указываем свойство `strikes` и присваиваем ему значение `100`, чтобы оно было больше, чем изначально.

Примечание. Инкрементирование исходного значения состояния.

Как вы заметили, вы будете часто менять исходное значение состояния на обновленное. Мы получаем исходное значение состояния, вызывая метод `this.state.strikes`. По причинам, связанным с производительностью, React может принять решение о периодическом пакетном обновлении состояния. Это может привести к тому, что исходное значение, хранящееся в этом состоянии, не будет синхронизироваться с реальностью. Чтобы помочь в этом, метод `setState` дает вам доступ к предыдущему объекту состояния через аргумент `prevState`. Используя этот аргумент, код мог бы выглядеть так:

```
this.setState((prevState) => {
  return {
    strikes: prevState.strikes + 100
  };
});
```

Конечный результат схож с нашим примером. Значение свойства `strikes` увеличивается на 100. Единственное значимое изменение заключается в гарантии, что значение свойства `strikes` будет тем, которое сохранено в объекте `State`.

Итак, следует ли использовать этот подход для обновления состояния? Есть весомые аргументы с обеих сторон. С одной стороны, правильно, так как метод `this.state` отлично подходит для большинства случаев. С другой стороны, рекомендуется лаконичность кода без введения дополнительных сложностей. Здесь нет однозначного ответа, поэтому используйте тот подход, который предпочитаете. Я все это рассказываю, потому что на практике вы можете столкнуться с разными подходами в написании кода.

Вам нужно сделать еще кое-что. Функция `timerTick` добавлена в компонент, но ее содержимое не имеет контекста, установленного для нашего компонента. Другими словами, это ключевое слово,

с помощью которого мы обращаемся к `setState`, вернет в текущей ситуации `TypeError`. Доступно несколько решений, каждое из которых немного смущает. Мы рассмотрим эту проблему подробно позднее. На данный момент мы намеренно привязываем функцию `timerTick` к компоненту, чтобы все ссылки `this` работали корректно. Добавьте следующую строку в код конструктора:

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

  this.state = {
    strikes: 0
  };
  this.timerTick = this.timerTick.bind(this);
}
```

Когда вы это сделаете, функция `timerTick` станет полезной нагрузкой компонента.

Рендеринг смены состояния

Если вы сейчас запустите свое приложение, то увидите, что значение `strikes` начинает увеличиваться на 100 с каждой секундой (см. рис. 8.2).

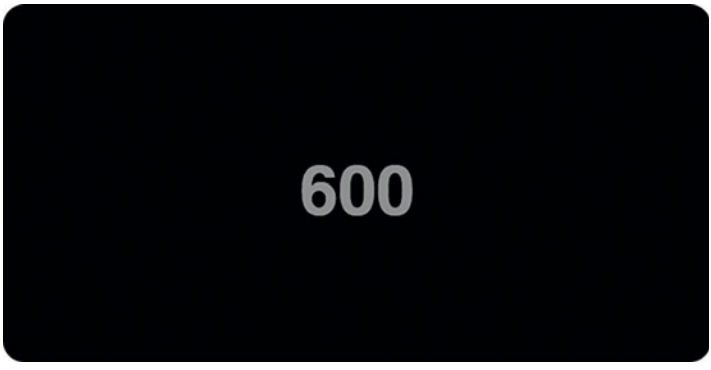


Рис. 8.2. Значение strikes увеличивается на 100 каждую секунду

Давайте на мгновение забудем, что происходит с кодом. Там нет никаких сложностей. Интересно, что все, что мы сделали, заканчивается обновлением вывода на экран. Это обновление связано с поведением React: *всякий раз, когда вы вызываете метод* `setState` *и обновляете что-либо в объекте состояния, метод* `render` *компонента вызывается автоматически*. Так последовательно вызывается метод `render` любого компонента. В финале вы видите на экране результат последней визуализации состояния пользовательского интерфейса приложения. Синхронизировать хранение данных и пользовательский интерфейс — одна из самых сложных проблем при разработке пользовательского интерфейса, поэтому приятно, что React позаботился об этом за нас.

Дополнительно: полный код

Все, что мы сейчас имеем, — это счетчик, который увеличивается на 100 в секунду. Ничего особенного, но пример охватывает все, что касается состояний, которые я хотел вам объяснить. Если вы хотите сверить свой код с моим, я привел полный код содержимого элемента `script`:

```
class LightningCounter extends React.Component {
  constructor(props) {
    super(props);

    this.state = {
      strikes: 0
    };

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

  timerTick() {
    this.setState({
      strikes: this.state.strikes + 100
    });
  }
```

```
  componentDidMount() {
    setInterval(this.timerTick, 1000);
  }

  render() {
    var counterStyle = {
      color: "#66FFFF",
      fontSize: 50
    };

    var count = this.state.strikes.toLocaleString();

    return (
      <h1 style={counterStyle}>{count}</h1>
    );
  }
}

class LightningCounterDisplay extends React.Component {
  render() {
    var commonStyle = {
      margin: 0,
      padding: 0
    };

    var divStyle = {
      width: 250,
      textAlign: "center",
      backgroundColor: "#020202",
      padding: 40,
      fontFamily: "sans-serif",
      color: "#999999",
      borderRadius: 10
    };
```

```
    var textStyles = {
      emphasis: {
        fontSize: 38,
        …commonStyle
      },

      smallEmphasis: {
        …commonStyle
      },

      small: {
        fontSize: 17,
        opacity: 0.5,
        …commonStyle
      }
    };

    return (
        <div style={divStyle}>
          <LightningCounter />
          <h2 style={textStyles.smallEmphasis}>ВСПЫШКИ
                МОЛНИИ</h2>
          <h2 style={textStyles.emphasis}>НА ПЛАНЕТЕ</h2>
          <p style={textStyles.small}>(с момента запуска
                кода)</p>
        </div>
    );
  }
}

ReactDOM.render(
  <LightningCounterDisplay />,
  document.querySelector("#container")
);
```

Если вы запустите этот код, вы увидите наш счетчик молний во всей красе. Пока вы на него любуетесь, найдите время и желание, чтобы просмотреть код еще раз и убедиться, что все строки вам понятны.

Заключение

Если не вдаваться в подробности, то мы разобрались, как создавать компоненты с состояниями. Мы лишь использовали таймер для обновления данных в объекте состояния. По-настоящему интересно будет тогда, когда мы начнем комбинировать взаимодействие пользователя с состояниями. До сих пор мы избегали действий с помощью мыши, нажатий клавиш клавиатуры и других связанных вещей, с которыми компоненты будут вступать во взаимодействие. В следующей главе мы займемся этим. По мере продвижения вперед вы начнете воспринимать состояния на совершенно новом для себя уровне. Надеюсь, вам понравится.

> Примечание: Если у вас возникнут какие-либо проблемы, задайте вопрос!
>
> Если у вас есть какие-либо вопросы или код не работает ожидаемым образом, не стесняйтесь задать вопрос! Опубликуйте свой вопрос на форуме **forum.kirupa.com** и получите помощь от самых дружелюбных и знающих людей в Интернете!

Глава 9
Переход от данных к пользовательскому интерфейсу

Когда вы разрабатываете приложения, свойства, состояния, компоненты, разметка JSX, методы `render` и другие термины React могут быть последним, о чем вы задумаетесь. В большинстве случаев вы будет иметь дело с данными в виде объектов JSON, массивов и других структур, а также с визуальными элементами. Преодоление пропасти между данными и тем, что вы в конечном итоге видите, может быть очень сложным! Не волнуйтесь. Эта глава поможет избавиться от таких неприятных ощущений, после того как мы рассмотрим некоторые общие сценарии.

Пример

Чтобы охватить все, что вам предстоит узнать, нам нужен пример. Он не слишком сложный. Создайте HTML-документ со следующим содержимым:

```
<! DOCTYPE html>
<html>

<head>
  <meta charset="utf-8">
  <title>От данных к интерфейсу</title>
```

```
  <script src="https://unpkg.com/react@16/umd/react.
        development.js"></script>
  <script src="https://unpkg.com/react-dom@16/umd/react-dom.
        development.js"></script>
  <script src="https://unpkg.com/babel-standalone@6.15.0/
        babel.min.js"></script>

  <style>
    #container {
      padding: 50px;
      background-color: #FFF;
    }
  </style>
</head>

<body>
  <div id="container"></div>

  <script type="text/babel">
    class Circle extends React.Component {
      render() {
        var circleStyle = {
          padding: 10,
          margin: 20,
          display: "inline-block",
          backgroundColor: this.props.bgColor,
          borderRadius: "50%",
          width: 100,
          height: 100,
        };

        return (
          <div style={circleStyle}>
          </div>
        );
```

```
        }
      }

      ReactDOM.render(
        <div>
          <Circle bgColor="#F9C240" />
        </div>,
        document.querySelector("#container");
      );
    </script>
  </body>
  </html>
```

Когда документ будет создан, откройте его в браузере и просмотрите, что у вас получилось. Если все пошло как по маслу, вас встретит замечательный желтый круг (см. рис. 9.1).

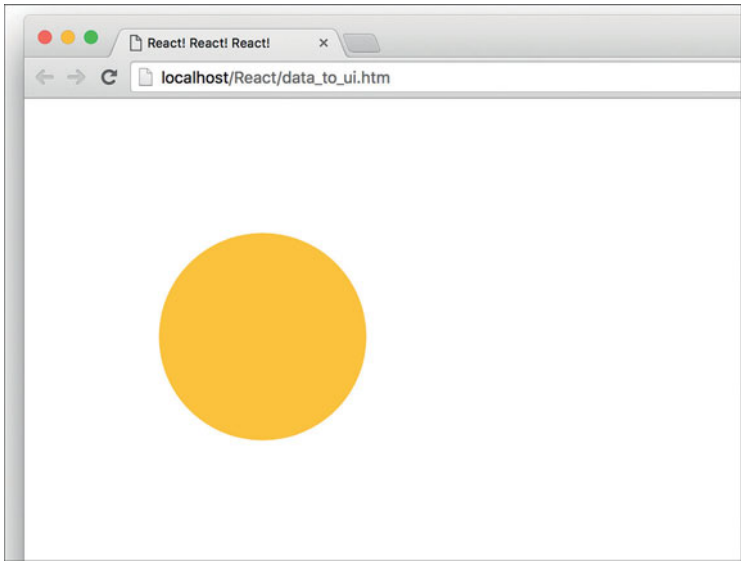


Рис. 9.1. Если все будет хорошо, вас встретит замечательный желтый круг

Если вы видите то, что вижу я, отлично! Теперь рассмотрим этот пример. Основная часть того, что вы видите, поступает из компонента `Circle`:

```
class Circle extends React.Component {
  render() {
    var circleStyle = {
      padding: 10,
      margin: 20,
      display: "inline-block",
      backgroundColor: this.props.bgColor,
      borderRadius: "50%",
      width: 100,
      height: 100,
    };

    return (
      <div style={circleStyle}>
      </div>
    );
  }
}
```

Он в основном состоит из объекта `circleStyle`, который содержит встроенные свойства стиля, превращающие скучный контейнер `div` в удивительный круг. Все настройки стилей жестко закодированы, за исключением свойства `backgroundColor`, которое берет свое значение из переменной `bgColor`.

За пределами компонента мы в итоге показываем круг с помощью обычного метода `ReactDOM.render`:

```
ReactDOM.render(
  <div>
    <Circle bgColor="#F9C240"/>
  </div>,
```

```
  destination
);
```

У нас есть один экземпляр объявленного компонента `Circle`, и мы объявляем его с помощью свойства `bgColor` со значением желаемого цвета. Теперь, когда компонент `Circle` определяется внутри метода `render`, мы слегка ограничены, особенно если будем иметь дело с данными, способными повлиять на компонент `Circle`. В следующих разделах мы рассмотрим, как решается эта проблема.

JSX-код может быть где угодно, часть II

Из главы 7 вы узнали, что JSX фактически может находиться вне функции `render` и использоваться как значение, присвоенное переменной или свойству. Например, мы можем бесстрашно сделать что-то вроде этого:

```
var theCircle = <Circle bgColor="#F9C240" />;

ReactDOM.render(
  <div>
    {theCircle}
  </div>,
  destination
);
```

Переменная `theCircle` хранит JSX-код для создания экземпляра компонента `Circle`. Инициализация этой переменной внутри функции `ReactDOM.render` приводит к отображению круга. Конечный результат ничем не отличается от того, что мы имели раньше, но освобождение экземпляра компонента `Circle` из оков метода `render` дает нам больше возможностей делать сумасшедшие и классные вещи.

Например, вы можете создать функцию, которая возвращает компонент `Circle`:

```
function showCircle() {
  var colors = ["#393E41", "#E94F37", "#1C89BF", "#A1D363"];
  var ran = Math.floor(Math.random() * colors.length);

  // возвращает круг с рандомным цветом
  return <Circle bgColor={colors[ran]} />;
}
```

В этом случае функция `showCircle` возвращает компонент `Circle` со свойством `bgColor`, которому присвоено случайное значение цвета. Чтобы в коде использовать функцию `showCircle`, все, что вам нужно сделать, это обработать ее внутри метода `ReactDOM.render`:

```
ReactDOM.render(
  <div>
    {showCircle()}
  </div>,
  destination
);
```

Пока обрабатываемое выражение возвращает JSX-код, вы можете поместить в скобки `{}` почти все, что угодно. Эта гибкость реально радует, потому что вы сможете сделать гораздо больше, если JavaScript-код находится вне функции `render`.

Работа с массивами

Теперь мы добрались до забавного раздела! Когда нужно отобразить несколько компонентов, вы не всегда можете вручную указать их:

```
ReactDOM.render(
  <div>
    {showCircle()}
    {showCircle()}
    {showCircle()}
```

```
    </div>,
    destination
);
```

Во многих реальных сценариях количество отображаемых компонентов связано с количеством элементов в массивах или массивоподобных структурах, с которыми вы работаете. Это вызывает некоторые осложнения. Например, предположим, что у нас есть массив `colors`, который выглядит следующим образом:

```
var colors = ["#393E41", "#E94F37", "#1C89BF", "#A1D363",
              "#85FFC7", "#297373", "#FF8552", "#A40E4C"];
```

Нам нужно создать компонент `Circle` для каждого элемента в этом массиве (и установить значение свойства `bgColor` для каждого элемента массива). Мы можем сделать это, создав массив компонентов `Circle`:

```
var colors = ["#393E41", "#E94F37", "#1C89BF", "#A1D363",
              "#85FFC7", "#297373", "#FF8552", "#A40E4C"];

var renderData = [];

for (var i = 0; i < colors.length; i++) {
  renderData.push(<Circle bgColor={colors[i]} />);
}
```

В этом листинге мы заполняем массив `renderData` компонентами `Circle` так же, как собирались изначально. Все идет по плану. React максимально упрощает отображение всех этих компонентов. Взгляните на выделенную строку. Это все, что нужно изменить:

```
ReactDOM.render(
  <div>
    {renderData}
```

```
    </div>,
    destination
);
```

Все, что мы сделали, это в методе `render` указали массив `renderData` как выражение, которое нужно обработать. Нам не нужно делать никаких других шагов, чтобы перейти от массива компонентов к представлению в браузере, типа показанного на рис. 9.2.

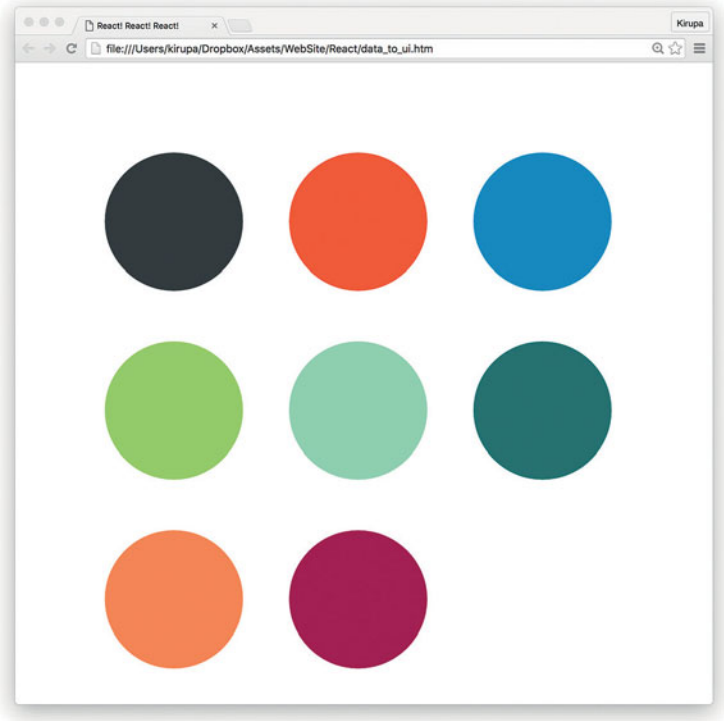


Рис. 9.2. Что вы должны увидеть в окне браузера

Ладно, я соврал. Нам нужно еще кое-что сделать, но это маленькая деталь. React обновляет пользовательский интерфейс очень быстрыми темпами, имея хорошее представление о том, что именно происходит в DOM. Он делает это несколькими способами, но один реально заметный момент — это внутреннее обозначение каждого элемента каким-то идентификатором.

Когда вы создаете элементы динамически (как мы это делаем с массивом компонентов `Circle`), эти идентификаторы не устанавливаются автоматически. Нам нужно сделать кое-что самостоятельно. Эта дополнительная работа заключается в установке свойства `key`, значение которого React использует для идентификации каждого конкретного компонента.

В нашем примере мы можем сделать следующее:

```
for (var i = 0; i < colors.length; i++) {
  var color = colors[i];
  renderData.push(<Circle key={i + color} bgColor={color} />);
}
```

Для каждого компонента мы указываем свойство `key` со значением в виде комбинации переменной `color` и индекса элемента массива `colors`. Это гарантирует, что каждый компонент, который мы динамически создаем, получает уникальный идентификатор, который React может затем использовать для оптимизации любых будущих обновлений пользовательского интерфейса.

> Проверь свою консоль, бро
>
> React реально хорош тем, что сообщает об ошибках, когда вы делаете что-то неправильно. Например, если вы создаете элементы или компоненты динамически и не указываете для них ключи, вы увидите предупреждение об этом в оболочке командной строки:
>
> ```
> Warning: Each child in an array or iterator should have a
> unique "key" prop.
> Check the top-level render call using .
> ```
>
> Когда вы работаете с React, рекомендуется периодически проверять оболочку командной строки на наличие любых сообщений. Даже если все работает нормально, ошибки могут быть скрытыми.

Заключение

Все советы и рекомендации, которые вы видели в этой главе, стали возможны благодаря одной вещи: *JSX — это JavaScript*. Благодаря этому вы можете использовать JSX везде, где допустим JavaScript. Иногда кажется, что мы делаем что-то совершенно странное, когда мы указываем нечто подобное:

```
for (var i = 0; i < colors.length; i++) {
  var color = colors[i];
  renderData.push(<Circle key={i + color} bgColor={color} />);
}
```

Несмотря на то, что мы передаем фрагменты JSX-кода в массив, все в результате работает магическим образом, когда `renderData` обрабатывается внутри метода `render`. В конечном счете приложение в браузере выглядит так:

```
for (var i = 0; i < colors.length; i++) {
  var color = colors[i];

  renderData.push(React.createElement(Circle,
  {
    key: i + color,
    bgColor: color
  }));
}
```

Так как JSX-код преобразуется в чистый JavaScript, все вновь обретает смысл. Благодаря этому мы можем не использовать JSX-код во всевозможных неудобных для нас сценариях, но по-прежнему можем получить желаемый результат. В конце концов, это всего лишь JavaScript.

Примечание: Если у вас возникнут какие-либо проблемы, задайте вопрос!

Если у вас есть какие-либо вопросы или код не работает ожидаемым образом, не стесняйтесь задать вопрос! Опубликуйте свой вопрос на форуме **forum.kirupa.com** и получите помощь от самых дружелюбных и знающих людей в Интернете!

Глава 10
События в React

До сих пор большинство наших примеров функционировали только при загрузке страницы. Как вы, наверное, догадались, это не очень хорошо. В большинстве приложений, особенно со сложными пользовательскими интерфейсами, которые вы будете создавать, можно выполнять массу действий в ответ на некие события. Эти события могут быть вызваны щелчком мыши, нажатием клавиши, изменением размера окна или целым рядом других жестов и взаимодействий. **События** — это связующее звено, делающее все это возможным.

Теперь вы, вероятно, знаете все о событиях из своего опыта, используя их в мире DOM. (Если это не так, я предлагаю сначала посетить веб-страницу для быстрого знакомства с событиями **www.kirupa.com/html5/javascript_events.htm**.) Способ, с помощью которого React обрабатывает события, немного отличается от JavaScript, и эти различия могут вас удивить, но не волнуйтесь, у вас есть эта книга! Мы начнем с нескольких простых примеров, а затем постепенно рассмотрим более сложные и интересные приложения.

Определение и реагирование на события

Самый простой способ познакомиться с событиями React — это использовать их в работе, и именно это вы сейчас и сделаете. В демонстрационных целях возьмем простой пример, состоящий из счетчика, который увеличивается каждый раз, когда вы нажимаете кнопку. Первоначально пример будет выглядеть так, как показано на рис. 10.1.

Рис. 10.1. Пример приложения

Каждый раз, когда вы нажимаете кнопку «плюс», значение счетчика увеличивается на 1. После того, как вы нажмете кнопку «плюс» несколько раз, он будет выглядеть примерно так, как показано на рис. 10.2.

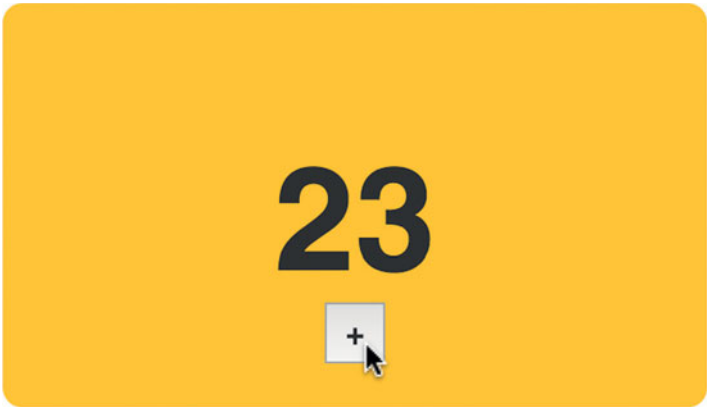


Рис. 10.2. После нажатия на кнопку «плюс» несколько раз

Судя по иллюстрациям, этот пример довольно прост. Каждый раз, когда вы нажимаете кнопку, происходит увеличение значения. Мы определяем это событие и создаем всевозможные действия React, чтобы обновить счетчик, когда это событие будет определено.

Начало работы

Чтобы сэкономить время, мы не будем создавать приложение с нуля. К настоящему моменту вы знаете, как работать с компонентами, стилями, состояниями и т. д. Поэтому мы начнем с частично реализованного примера, который содержит все необходимое, кроме функций, связанных с событиями.

Сначала создайте HTML-документ и добавьте в него следующий код:

```
<! DOCTYPE html>
<html>

<head>
  <meta charset="utf-8">
  <title>События</title>
  <script src="https://unpkg.com/react@16/umd/react.
        development.js"></script>
  <script src="https://unpkg.com/react-dom@16/umd/react-dom.
        development.js"></script>
  <script src="https://unpkg.com/babel-standalone@6.15.0/
        babel.min.js"></script>
  <style>
    #container {
      padding: 50px;
      background-color: #FFF;
    }
  </style>
</head>

<body>
  <div id="container"></div>
  <script type="text/babel">

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

HTML-документ готов, и теперь пора добавить частично реализованный пример счетчика. Внутри элемента `script` под контейнером `div` вставьте следующий код:

```
class Counter extends React.Component {
  render() {
    var textStyle = {
      fontSize: 72,
      fontFamily: "sans-serif",
      color: "#333",
      fontWeight: "bold"
    };

    return (
      <div style={textStyle}>
      {this.props.display}
    </div>
    );
  }
}

class CounterParent extends React.Component {
  constructor(props) {
    super(props);

    this.state = {
      count: 0
    };
  }

  render() {
    var backgroundStyle = {
      padding: 50,
      backgroundColor: "#FFC53A",
      width: 250,
```

```
        height: 100,
        borderRadius: 10,
        textAlign: "center"
      };

      var buttonStyle = {
        fontSize: "1em",
        width: 30,
        height: 30,
        fontFamily: "sans-serif",
        color: "#333",
        fontWeight: "bold",
        lineHeight: "3px"
      };

      return (
        <div style={backgroundStyle}>
          <Counter display={this.state.count} />
          <button style={buttonStyle}>+</button>
        </div>
      );
    }
  }

  ReactDOM.render(
    <div>
      <CounterParent />
    </div>,
    document.querySelector("#container")
  );
```

Теперь просмотрите приложение в своем браузере, чтобы убедиться, что оно работает. Вы увидите основу счетчика. Прекратите читать на пару минут, чтобы проанализировать набранный код. Все должно быть знакомо. Единственное, при нажатии кнопки

«плюс», ничего не происходит. Мы исправим это в следующем разделе.

Создание кнопки действия

Каждый раз, когда мы нажимаем кнопку «плюс», нужно, чтобы значение счетчика увеличилось на 1. Для этого следует поступить примерно так:

1. Определить событие щелчка на кнопке.

2. Внедрить обработчик событий, чтобы реагировать на щелчок и увеличивать значение свойства `this.state.count`, на которое опирается счетчик.

Мы перейдем к списку, начиная с определения события `click`. В React вы определяете событие, указав все необходимое в JSX-коде. Если конкретнее, *вы указываете в разметке как слушаемое событие, так и обработчик событий, который будет вызван*. Чтобы сделать это, найдите функцию `return` в коде компонента `CounterParent` и внесите следующие изменения (выделены цветом):

```
    .
    .
    .
  return (
    <div style={backgroundStyle}>
      <Counter display={this.state.count}/>
      <button onClick={this.increase} style={buttonStyle}>+
          </button>
    </div>
  );
```

Мы инструктировали React вызвать функцию `increase`, когда происходит событие `onClick`. Теперь реализуем функцию `increase`

(обработчик событий). Внутри компонента `CounterParent` внесите следующие изменения (выделены цветом):

```
class CounterParent extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      count: 0
    };

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

  increase(e) {
    this.setState({
      count: this.state.count + 1
    });
  }

  render() {
    var backgroundStyle = {
      padding: 50,
      backgroundColor: "#FFC53A",
      width: 250,
      height: 100,
      borderRadius: 10,
      textAlign: "center"
    };

    var buttonStyle = {
      fontSize: "1em",
      width: 30,
      height: 30,
      fontFamily: "sans-serif",
      color: "#333",
```

```
        fontWeight: "bold",
        lineHeight: "3px"
      };

      return (
        <div style={backgroundStyle}>
          <Counter display={this.state.count} />
          <button onClick={this.increase} style={buttonStyle}>+
              </button>
        </div>
      );
    }
}
```

В этих строках каждый вызов функции `increase` увеличивает значение свойства `this.state.count` на 1. Поскольку мы имеем дело с событиями, функция `increase` (в качестве назначенного обработчика события) получит доступ к любым аргументам события. Мы установили эти аргументы для доступа к `e`, как это видно из сигнатуры функции `increase` (то есть как выглядит ее объявление). Мы немного поговорим о различных событиях и их свойствах, когда подробно рассмотрим события. Наконец, в конструкторе мы привязываем значение к функции `increase`.

Теперь давайте посмотрим в браузере, что получилось. Как только приложение загрузится, нажмите кнопку «плюс», чтобы увидеть код в действии. Значение счетчика должно увеличиваться с каждым щелчком. Разве это не удивительно?

Свойства события

Как известно, события передают обработчикам событий так называемые **аргументы событий**. Они содержат кучу свойств, уникальных для каждого типа событий. В обычном DOM событие имеет свой собственный тип. Например, если вы имеете дело с событием мыши, событие и его аргументы имеют тип `MouseEvent`. Этот объект

`MouseEvent` позволяет вам получить доступ к данным, относящимся к мыши, например, о том, какая кнопка была нажата или в какой позиции экрана. Аргументы событий, связанных с нажатием клавиш на клавиатуре, имеют тип `KeyboardEvent`. Объект `KeyboardEvent` содержит свойства, которые (среди многих других) позволяют вам определить, какая клавиша была нажата. Я мог бы продолжать вечно для каждого другого типа события, но вы поняли суть. События каждого типа содержат свой собственный набор свойств, к которым вы можете получить доступ через обработчик выбранного события.

Синтетические события

В React, когда вы указываете событие с помощью JSX-кода, как мы это делали с `onClick`, вы не обращаетесь непосредственно к обычным событиям DOM. Вместо этого вы имеете дело со специфичным для React событием, известным под названием `SyntheticEvent`. Обработчики событий не получают аргументы обычного события типа `MouseEvent`, `KeyboardEvent` и т. д. Они всегда получают аргументы типа `SyntheticEvent`, которые оборачивают собственные события браузера. Как это отражается в коде? На удивление изменений мало.

Каждое событие `SyntheticEvent` содержит следующие свойства:

```
boolean bubbles
boolean cancelable
DOMEventTarget currentTarget
boolean defaultPrevented
number eventPhase
boolean isTrusted
DOMEvent nativeEvent
void preventDefault()
boolean isDefaultPrevented()
void stopPropagation()
boolean isPropagationStopped()
```

```
DOMEventTarget target
number timeStamp
string type
```

Эти свойства кажутся довольно простыми! Зависит от того, событие какого типа содержится в обертке `SyntheticEvent`. Это означает, что событие `SyntheticEvent`, обернувшее событие `MouseEvent`, будет иметь доступ к характерным для мыши свойствам, таким как:

```
boolean altKey
number button
number buttons
number clientX
number clientY
boolean ctrlKey
boolean getModifierState(key)
boolean metaKey
number pageX
number pageY
DOMEventTarget relatedTarget
number screenX
number screenY
boolean shiftKey
```

Аналогично, событие `SyntheticEvent`, обернувшее событие `KeyboardEvent`, будет иметь доступ к этим дополнительным свойствам, связанным с клавиатурой:

```
boolean altKey
number charCode
boolean ctrlKey
boolean getModifierState(key)
string key
number keyCode
```

```
string locale
number location
boolean metaKey
boolean repeat
boolean shiftKey
number which
```

В конце концов все это означает, что вы по-прежнему получаете такую же функциональность с помощью события `SyntheticEvent`, что и в классическом мире DOM.

Теперь постарайтесь кое-что усвоить: *не обращайтесь к традиционной документации по событиям DOM при использовании событий `SyntheticEvent` и их свойств*. Поскольку событие `SyntheticEvent` содержит собственное событие DOM, события и их свойства могут не сопоставляться друг с другом. Некоторые события DOM даже не существуют в React. Чтобы избежать проблем, если вы хотите глубоко изучить события `SyntheticEvent` и любые их свойства, *обратитесь к документу System React Event System* (**facebook.github.io/react/docs/events.html**).

Работа со свойствами события

Теперь вы довольно много знаете о DOM и событиях `SyntheticEvent`. Чтобы закрепить знания, давайте напишем код и применим полученную теорию на практике. Сейчас счетчик увеличивается на 1 каждый раз, когда вы нажимаете кнопку «плюс». Мы хотим *увеличивать счетчик на 10 при удерживании клавиши **Shift** на клавиатуре* и щелчке по кнопке «плюс» мышью.

Мы можем это сделать, используя свойство `shiftKey`, которое существует в событии `SyntheticEvent` при использовании мыши:

```
boolean altKey
number button
number buttons
number clientX
```

```
number clientY
boolean ctrlKey
boolean getModifierState(key)
boolean metaKey
number pageX
number pageY
DOMEventTarget relatedTarget
number screenX
number screenY
boolean shiftKey
```

Способ работы этого свойства прост. Если нажата клавиша `Shift` при срабатывании данного события мыши, значение свойства `shiftKey` равно `true`. В противном случае значение свойства `shiftKey` будет `false`. Чтобы увеличить счетчик на 10 при нажатии клавиши **Shift**, вернитесь к функции `increase` и внесите следующие изменения (выделены цветом) в код:

```
increase(e) {
  var currentCount = this.state.count;

if (e.shiftKey) {
  currentCount += 10;
  } else {
    currentCount += 1;
  }

  this.setState({
    count: currentCount
  });
}
```

Когда вы внесете изменения, просмотрите приложение в браузере. Каждый раз, когда вы нажимаете кнопку «плюс», счетчик будет увеличиваться на 1, как и до этого. Если вы нажмете кнопку «плюс»,

удерживая нажатой клавишу **Shift**, обратите внимание, что счетчик увеличивается на 10.

Все получилось, потому что мы меняем инкрементируемое значение в зависимости от того, нажата ли клавиша **Shift**. За это отвечают следующие строки:

```
if (e.shiftKey) {
  currentCount += 10;
} else {
  currentCount += 1;
}
```

Если свойство `shiftKey` в аргументе события `SyntheticEvent` истинно (`true`), мы увеличиваем счетчик на `10`. Если значение `shiftKey` ложно(`false`) — то на `1`.

Действия с событиями

Мы еще не закончили! До этого момента мы рассмотрели, как работать с событиями в React, но на поверхностном уровне. В реальном мире редко будет все так просто. Реальные приложения будут более сложными, и, поскольку React позволяет достигать результата разными путями, вам нужно будет изучить некоторые дополнительные трюки и методы, связанные с событиями, чтобы приложения заработали. Этот раздел главы вам как раз и поможет. Мы рассмотрим некоторые распространенные ситуации, с которыми вы можете столкнуться, и способы работы с ними.

Вы не можете напрямую прослушивать события компонентов

Ваш компонент — это не что иное, как кнопка или другой элемент пользовательского интерфейса, с которым пользователи будут взаимодействовать. Вы не можете сделать нечто такое, что указано в выделенной строке:

```
class CounterParent extends React.Component {
  constructor(props) {
    super(props);

    this.state = {
      count: 0
    };

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

  increase(e) {
    this.setState({
      count: this.state.count + 1
    });
  }

  render() {
    return (
      <div>
        <Counter display={this.state.count} />
        <PlusButton onClick={this.increase} />
      </div>
    );
  }
}
```

На первый взгляд выделенная строка JSX-кода выглядит абсолютно корректно. Когда кто-то щелкает мышью по компоненту `PlusButton`, вызывается функция `increase`. Если вам интересно, ниже показан код компонента `PlusButton`:

```
class PlusButton extends React.Component {
  render() {
    return (
```

```
        <button>
        +
        </button>
      );
    }
}
```

Компонент `PlusButton` не делает ничего необычного; он возвращает только один HTML-элемент.

Неважно, как вы используете это событие. Неважно, насколько прост или очевиден HTML-код, который мы возвращаем с помощью компонента. *Вы не можете напрямую прослушивать события компонентов.* Это связано с тем, что компоненты — это «обертка» для элементов DOM. Что еще означает прослушивание события компонента? Когда компонент разворачивается в элементы DOM, внешний HTML-элемент действует как нечто, что вы слушаете как событие? Это какой-то другой элемент? Как вы различаете прослушивание события и объявление события, которое прослушиваете?

На все эти вопросы нет четкого ответа. Можно только сказать, что решение состоит в том, чтобы не прослушивать события компонентов. К счастью, есть обходной путь: мы можем использовать обработчик событий как свойство и передать его компоненту. Внутри компонента мы можем затем назначить событие элементу DOM и присвоить обработчику события значение переданного свойства. Возможно, вам непонятно, поэтому рассмотрим пример.

Взгляните на следующую выделенную строку:

```
class CounterParent extends React.Component {
  .
  .
  .
  render() {
    return (
      <div>
```

```
          <Counter display={this.state.count} />
          <PlusButton clickHandler={this.increase} />
        </div>
      );
    }
  }
```

В этом примере мы создаем свойство `clickHandler`, значение которого — обработчик события `increase`. Внутри компонента `PlusButton` мы можем написать такой код:

```
class PlusButton extends React.Component {
  render() {
    return (
      <button onClick={this.props.clickHandler}>
      +
      </button>
    );
  }
}
```

В элементе `button` мы указываем событие `onClick` и присваиваем его значение свойству `clickHandler`. Во время выполнения свойство обрабатывается как функция `increase`, а нажатие кнопки «плюс» гарантирует вызов функции `increase`. Так наша проблема решается, позволяя компоненту участвовать во всем этом событии.

Прослушивание стандартных событий DOM

Если вы считаете, что предыдущий раздел был очень сложным, подождите, вы пока не видели, что есть еще. Не все события DOM имеют эквивалентные события `SyntheticEvent`. Можно подумать, что достаточно просто добавить префикс `on` и извлечь событие, которое вы определяете, встроив его в JSX-код:

```
class Something extends React.Component {
  .
  .
  .
  handleMyEvent(e) {
    // что-то сделать
  }

  render() {
    return (
      <div onSomeEvent={this.handleMyEvent}>Привет!</div>
    );
  }
}
```

Так не получится! Для событий, которые React официально не поддерживает, вы должны следовать традиционному подходу, применяя слушатель `addEventListener` с несколькими дополнительными связями для перехода.

Взгляните на следующий фрагмент кода:

```
class Something extends React.Component {
  .
  .
  .
  handleMyEvent(e) {
    // что-то сделать
  }

  componentDidMount() {
    window.addEventListener("someEvent", this.handleMyEvent);
  }

  componentWillUnmount() {
    window.removeEventListener("someEvent", this.handleMyEvent);
  }
```

```
  render() {
    return (
      <div>Привет!</div>
    );
  }
}
```

У нас есть компонент `Something`, который определяет событие с именем `someEvent`. Мы начинаем обрабатывать это событие по методу `componentDidMount`, который автоматически вызывается при визуализации компонента. Мы отслеживаем событие с помощью слушателя `addEventListener` и указываем как событие, так и обработчик событий для вызова.

Все это довольно просто. Единственное, что вам нужно иметь в виду, что требуется удалить слушатель событий, если компонент будет уничтожен. Для этого вы можете использовать противоположность метода `componentDidMount` — метод `componentWillUnmount`. Поместите вызов `removeEventListener` внутри этого метода, чтобы гарантировать отсутствие следов слушателя событий после уничтожения компонента.

Значение `this` внутри обработчика событий

Когда вы имеете дело с событиями в React, значение `this` в коде обработчика событий отличается от того, что вы обычно видите в DOM, не относящейся к React. В этом случае значение `this` внутри обработчика события относится к элементу, который активировал событие:

```
function doSomething(e) {
  console.log(this); // button element
}

var foo = document.querySelector("button");
foo.addEventListener("click", doSomething, false);
```

В React значение `this` не относится к элементу, который активировал событие. Значение крайне бесполезно (но корректно) и неопределимо. Вот почему нам нужно явно указать связь с помощью метода `bind`, как показано в листинге ниже:

```
class CounterParent extends React.Component {
  constructor(props) {
    super(props);

    this.state = {
      count: 0
    };

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

  increase(e) {
    console.log(this);

    this.setState({
      count: this.state.count + 1
    });
  }

  render() {
    return (
      <div>
        <Counter display={this.state.count} />
        <button onClick={this.increase}>+</button>
      </div>
    );
  }
}
```

В этом примере значение `this` внутри обработчика события увеличения относится к компоненту `CounterParent`. Оно не относится к элементу, вызвавшему событие. Вы можете указать такое поведение, связав значение `this` с компонентом внутри конструктора.

React… Почему? Почему?

Прежде чем продолжить, поговорим немного о том, почему в React решили отказаться от ранее использовавшегося способа. На то было две причины:

1. Совместимость с браузерами.

2. Повышение производительности.

Давайте немного поразмыслим над этим.

Совместимость с браузерами

Обработка событий отлично производится в современных браузерах, но если вы вернетесь к более старым версиям браузеров, ситуация резко ухудшается. Оборачивая собственные события в объекты типа `SyntheticEvent`, React освобождает вас от работы с обработчиком событий.

Повышение производительности

В сложных пользовательских интерфейсах чем больше обработчиков событий используется, тем больше памяти занимает приложение. Вручную управлять этим несложно, но слегка утомительно, если вы попытаетесь группировать события под общим предком. А иногда это невозможно. В некоторых случаях сложность сводит на нет все преимущества. В React это решается довольно интересно.

React никогда напрямую не прикрепляет обработчики событий к элементам DOM. *Он использует один обработчик событий в корне*

документа, который отвечает за прослушку всех событий и при необходимости вызывает соответствующий обработчик событий (см. рис. 10.3).

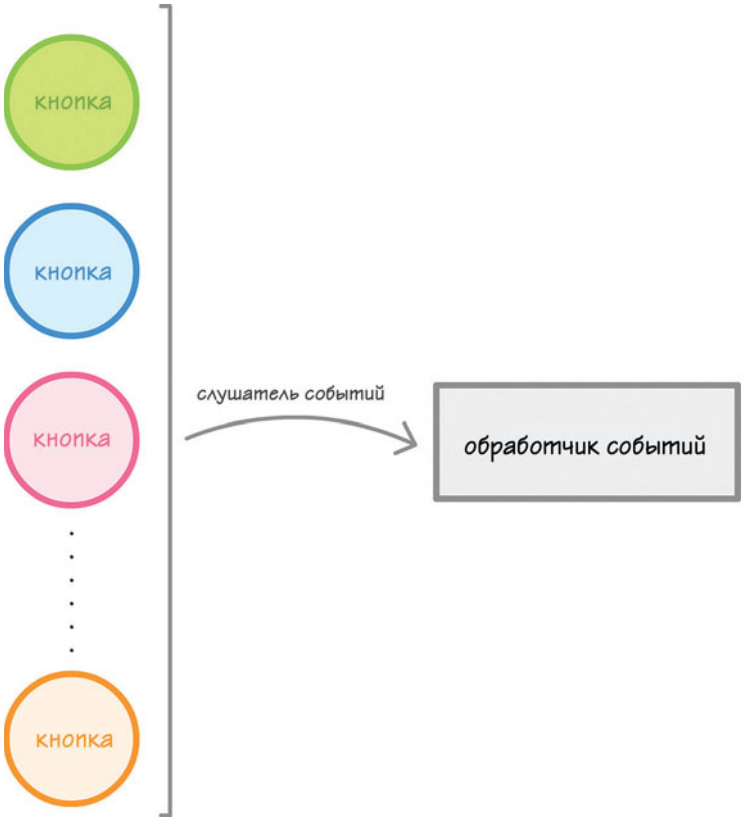


Рис. 10.3. React использует один обработчик событий в корне документа

Это освобождает вас от необходимости самостоятельно заниматься оптимизацией кода, связанного с обработчиками событий. Если вы занимались этим ранее, то теперь можете расслабиться, так как React позаботится об этой утомительной задаче. Если вам никогда не приходилось самостоятельно оптимизировать код, связанный с обработкой событий, считайте, вам повезло.

Заключение

Вы потратите много времени на изучение событий, и эта глава вывалила на вас очень много информации. Мы начали с основ прослушивания событий и обработки событий. К концу главы мы полностью во всем разобрались и изучили возможные ошибки, с которыми вы можете столкнуться, если не будете достаточно осторожны. Не повторяйте этих ошибок. Они опечалят.

> Примечание: Если у вас возникнут какие-либо проблемы, задайте вопрос!
>
> Если у вас есть какие-либо вопросы или код не работает ожидаемым образом, не стесняйтесь задать вопрос! Опубликуйте свой вопрос на форуме **forum.kirupa.com** и получите помощь от самых дружелюбных и знающих людей в Интернете!

Глава 11
Жизненный цикл компонента

Вначале мы кинули беглый взгляд на компоненты и на то, что они делают. Когда вы узнали больше о React, то пришли к выводу, что компоненты не так уж и просты. Они помогают взаимодействовать со свойствами, состояниями и событиями и часто отвечают за корректную работу других компонентов. Отслеживание всех компонентов бывает весьма трудоемким занятием.

Чтобы помочь в этом, React предоставляет методы жизненного цикла. **Методы жизненного цикла** — это специальные методы, которые автоматически вызываются компонентами. Они сообщают нам о важных вехах в жизни компонента, и мы можем использовать эти уведомления, чтобы обратить внимание или повлиять на то, что должен сделать компонент.

В этой главе мы рассмотрим методы жизненного цикла и поговорим о том, как с ними работать.

> Примечание: Все может измениться!
> Предлагаю рассмотреть изменения, касающиеся методов жизненного цикла. Материал, который вы прочитаете в этой главе, учитывает последние рекомендации, но обратите внимание, что эта информация может устареть или измениться. Перейдите по ссылке **bit.ly/lifecycleChanges**, чтобы быть в курсе последних событий.

Знакомство с методами жизненного цикла

Методы жизненного цикла не очень сложны. Вы можете представить их как великие обработчики событий, которые вызываются на разных этапах жизни компонента. Как и в случае с обработчиками событий, вы можете написать код, чтобы на этих этапах происходили какие-то действия. Прежде чем продолжить, пришло время быстро познакомиться с методами жизненного цикла:

- `componentWillMount`
- `componentDidMount`
- `componentWillUnmount`
- `componentWillUpdate`
- `componentDidUpdate`
- `shouldComponentUpdate`
- `componentWillReceiveProps`
- `componentDidCatch`

Это еще не все. Существует еще один метод, который, строго говоря, не является методом жизненного цикла: бесстыжий метод `render`.

Некоторые из этих имен, вероятно, вам знакомы, а некоторые вы видите в первый раз. Не волнуйтесь. К концу главы вы будете знакомы со всеми методами. Мы рассмотрим их с разных точек зрения, а начнем с демонстрационного кода.

Методы жизненного цикла в действии

Изучение методов жизненного цикла так же захватывает, как заучивание наизусть названий туристических достопримечательностей, которые вы никогда не планируете посетить. Чтобы вытерпеть, давайте познакомимся с методами с помощью простого примера, прежде чем на вас свалится вся эта сухая теория.

Чтобы воспроизвести пример, перейдите по URL-адресу **www.kirupa.com/react/lifecycle_example.htm**. Когда эта страница загрузится, вы увидите счетчик, который вам уже знаком (см. рис. 11.1).

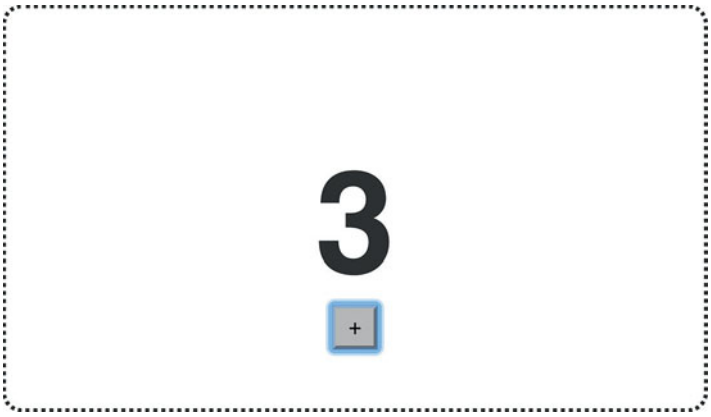


Рис. 11.1. Одна из вариаций счетчика

Не нажимайте кнопку, вообще ничего не делайте. (Если вы не утерпели, обновите страницу, чтобы начать пример с самого начала.) Я говорю это специально, а не просто так. Вы должны видеть эту страницу в первоначальном виде перед взаимодействием с ней.

Теперь откройте панель инструментов разработчика браузера и взгляните на вкладку **Console** (Консоль). В браузере Chrome вкладка выглядит примерно так, как показано на рис. 11.2.

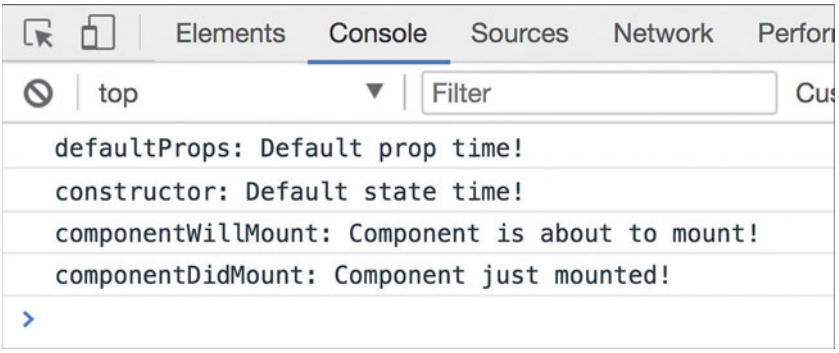


Рис. 11.2. Вид вкладки Console в браузере Chrome

Обратите внимание на вывод в консоли. Вы увидите сообщения, которые начинаются с имен методов жизненного цикла. Если

вы нажмете кнопку «плюс» сейчас, обратите внимание, что в консоли отображаются другие, вызванные методы жизненного цикла (см. рис. 11.3).

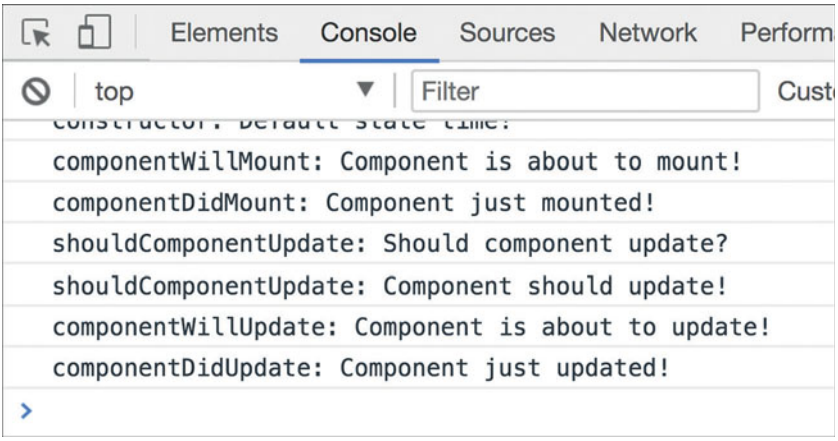


Рис. 11.3. Вызвано несколько методов жизненного цикла

Поэкспериментируйте с этим примером. Вы увидите, что можно разместить все эти методы жизненного цикла в контексте компонента, с которым вы уже знакомы. Когда вы нажимаете кнопку «плюс», выводятся записи о методах жизненного цикла. В конце концов, когда счетчик приблизится к значению 5, пример исчезнет, и в консоли появится запись: `componentWillUnmount: Component is about to be removed from the DOM!`: т. е. компонент будет удален из DOM! Теперь вы достигли конца этого примера. Конечно, чтобы начать заново, вы можете обновить страницу.

Теперь, когда вы видели пример целиком, давайте быстро рассмотрим компонент, который отвечает за его работу (полный источник: **github.com/kirupa/kirupa/blob/master/reactjs/lifecycle.htm**):

```
class CounterParent extends React.Component {
  constructor(props) {
    super(props);

    console.log("constructor: Default state time!");
```

```
    this.state = {
      count: 0
    };

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

  increase() {
    this.setState({
      count: this.state.count + 1
    });
  }

  componentWillUpdate(newProps, newState) {
    console.log("componentWillUpdate: Component is about
                 to update!");
  }

  componentDidUpdate(currentProps, currentState) {
    console.log("componentDidUpdate: Component just updated!");
  }

  componentWillMount() {
    console.log("componentWillMount: Component is about to
                 mount!");
  }

  componentDidMount() {
    console.log("componentDidMount: Component just mounted!");
  }

  componentWillUnmount() {
    console.log("componentWillUnmount: Component is about
                 to be removed from the DOM!");
  }
```

```
shouldComponentUpdate(newProps, newState) {
  console.log("shouldComponentUpdate: Should component
              update?");
  if (newState.count < 5) {
    console.log("shouldComponentUpdate: Component should
                 update!");
    return true;
  } else {
    ReactDOM.unmountComponentAtNode(destination);
    console.log("shouldComponentUpdate: Component should
                 not update!");
    return false;
  }
}

componentWillReceiveProps(newProps) {
  console.log("componentWillReceiveProps: Component will
              get new props!");
}

render() {
  var backgroundStyle = {
    padding: 50,
    border: "#3332px dotted",
    width: 250,
    height: 100,
    borderRadius: 10,
    textAlign: "center"
  };

  return (
    <div style={backgroundStyle}>
      <Counter display={this.state.count} />
      <button onClick={this.increase}>
      +
```

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

  console.log("defaultProps: Default prop time!");
  CounterParent.defaultProps = {

  };
```

Потратьте пару минут, чтобы понять, как работает весь этот код. Он кажется длинным, но основная его часть — это список методов жизненного цикла, определенных с помощью инструкций `console.log`. После того, как вы изучите код, поэкспериментируйте с примером еще раз. Доверьтесь мне. *Чем больше времени вы потратите на этот пример, чтобы разобраться, как он работает, тем проще будет усвоить дальнейший материал.* Следующие разделы будут ужасно скучными, когда мы будем рассматривать каждый метод жизненного цикла на этапах рендеринга, обновления и размонтирования. Я вас предупредил!

Этап начального рендеринга

Когда компонент начинает свою жизнь и отправляется в DOM, вызываются методы жизненного цикла, показанные на рис. 11.4.

То, что отображается в консоли при загрузке приложения, — текстовая версия счетчика. Теперь давайте рассмотрим, чем занимается каждый метод жизненного цикла.

Рис. 11.4. Сначала вызываются методы жизненного цикла

Получение свойств по умолчанию

Это свойство компонента позволяет указать дефолтное значение `this.props`. Если мы хотим настроить свойство `name` компонента `CounterParent`, код будет выглядеть следующим образом:

```
CounterParent.defaultProps = {
  name: "Терминатор"
};
```

Этот код выполняется до того, как компонент будет создан, или в него будут переданы свойства из родительских компонентов.

Получение состояния по умолчанию

Этот шаг выполняется внутри конструктора компонента. Вы можете указать дефолтное значение `this.state` в части создания компонента:

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

  console.log("constructor: Default state time!");

  this.state = {
    count: 0
  };

  this.increase = this.increase.bind(this);
}
```

Обратите внимание, как мы определяем объект `state` и инициализируем его с помощью свойства `count`, значение которого равно `0`.

componentWillMount

Это последний метод, который вызывается до того, как компонент будет передан в DOM. Здесь важно отметить: если вы вызываете метод `setState` внутри метода `componentWillMount`, компонент не будет повторно рендериться.

render

Этот метод вам очень хорошо знаком. Каждый компонент должен иметь этот метод, так как он отвечает за возврат JSX-кода. Если не нужно ничего рендерить (визуализировать), верните значение `null` или `false`.

`componentDidMount`

Этот метод вызывается сразу после того, как компонент визуализирован и помещен в DOM. На этом этапе вы можете безопасно выполнять любые операции запроса DOM, не беспокоясь о готовности компонента. Если у вас есть код, зависимый от готовности компонента, вы также можете указать его здесь.

За исключением метода `render`, все перечисленные методы жизненного цикла *могут срабатывать только один раз*. Это поведение сильно отличается от методов, которые вы увидите далее.

Этап обновления

После того как компоненты будут добавлены в DOM, они могут обновиться и повторно выполнить рендеринг при возникновении изменений свойств или изменений состояния. В это время вызывается другая коллекция методов жизненного цикла.

Изменение состояний

Во-первых, давайте взглянем на изменение состояния. Как упоминалось ранее, когда происходит изменение состояния, компонент вновь вызывает свой метод `render`. Любые компоненты, которые полагаются на вывод этого компонента, также вызывают методы `render`. Это делается для того, чтобы компонент всегда отображал самую последнюю версию. Все это верно, но это лишь часть информации о том, что происходит.

Когда происходит изменение состояния, вызываются все методы жизненного цикл, как показано на рис. 11.5.

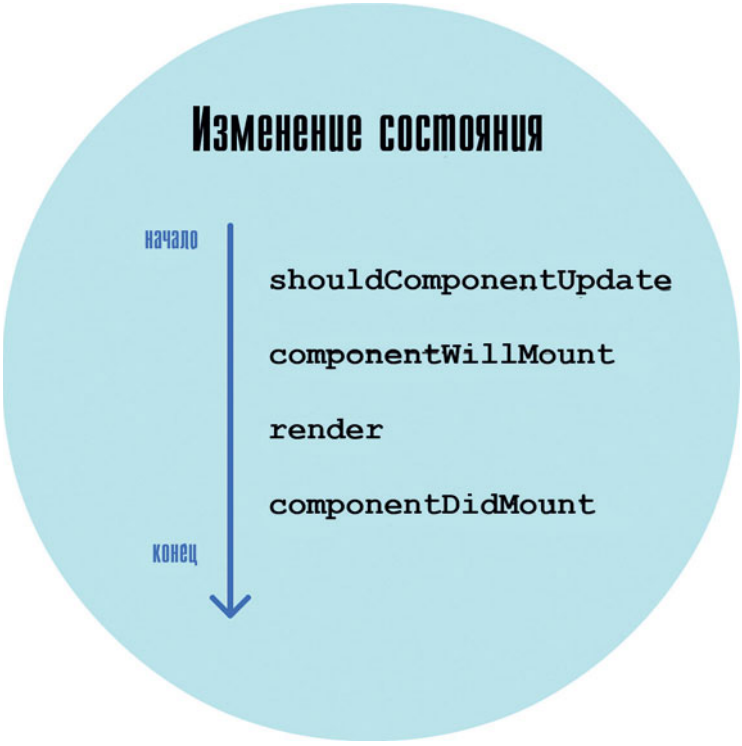


Рис. 11.5. Методы жизненного цикла, вызываемые при изменении состояния

Рассмотрим, для чего предназначены эти методы жизненного цикла:

shouldComponentUpdate

Иногда не нужно, чтобы компонент обновлялся при изменении состояния. Метод `shouldComponentUpdate` позволяет контролировать, выполнять ли обновление. Если вы используете этот метод со значением `true`, компонент будет обновляться. Если этот метод возвращает значение `false`, этот компонент пропускает обновление.

Вероятно, звучит немного запутанно, поэтому взгляните на простой код:

```
shouldComponentUpdate(newProps, newState) {
  console.log("shouldComponentUpdate: Should component
               update?");

  if (newState.count < 5) {
    console.log("shouldComponentUpdate: Component should
                 update!");
    return true;
  } else {
    ReactDOM.unmountComponentAtNode(destination);
    console.log("shouldComponentUpdate: Component should not
                 update!");
    return false;
  }
}
```

Этот метод вызывается с двумя аргументами с именами `newProps` и `newState`. В этом фрагменте кода проверяется, меньше ли или равно `5` значение свойства `state`. Если значение меньше или равно `5`, возвращается значение `true`, т. е. компонент должен обновляться. Если значение больше `5`, возвращается значение `false`, т. е. компонент не должен обновляться.

`componentWillUpdate`

Этот метод вызывается непосредственно перед обновлением компонента. Здесь ничего интересного не происходит. Следует отметить, что вы не можете изменить состояние, вызвав `this.setState` из этого метода.

`render`

Если вы не настроили поведение при обновлении с помощью метода `shouldComponentUpdate`, код метода `render` вновь вызывается, чтобы убедиться, что компонент отображается корректно.

componentDidUpdate

Этот метод вызывается после обновления компонента и после вызова метода `render`. Если вам нужно выполнить какой-либо код после обновления компонента, сейчас самое время.

Изменения свойств

В другой ситуации обновления компонента, значение его свойства изменяется после того, как оно было визуализировано в DOM. В этом случае вызовы наследуют методы жизненного цикла, как показано на рис. 11.6.

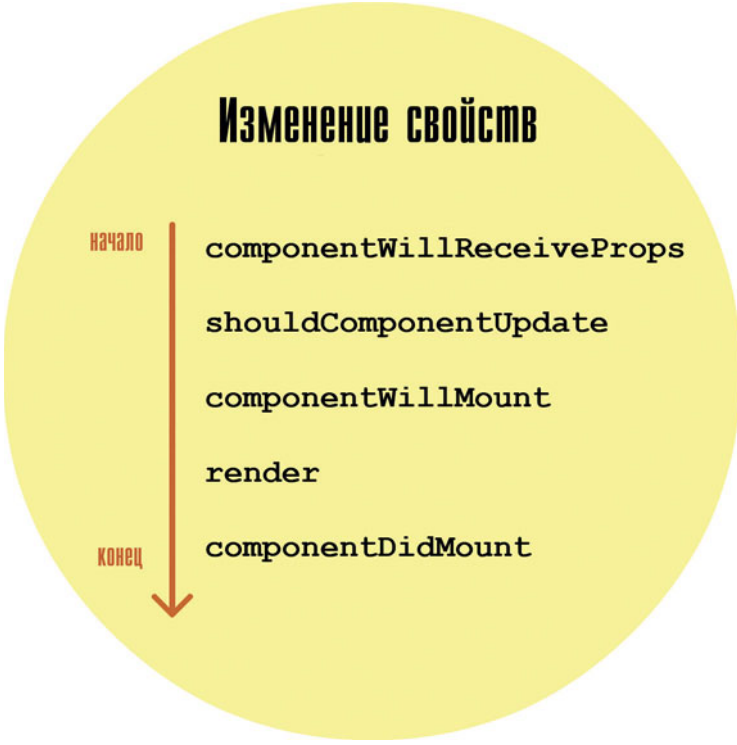


Рис. 11.6. Методы жизненного цикла, когда значение свойств компонента изменяется

Единственный новый метод — `componentWillReceiveProps`. Этот метод получает один аргумент, который содержит новое значение свойства для присваивания.

Вы уже видели остальные методы жизненного цикла при анализе изменения состояния, поэтому давайте не будем к ним возвращаться. Их поведение идентично, когда речь идет об изменении свойств.

Этап размонтирования

Последний этап, на который нужно обратить внимание, — это когда компонент уничтожается и удаляется из DOM (см. рис. 11.7).

Здесь активен только один метод жизненного цикла, который называется `componentWillUnmount`. На этом этапе выполняются задачи, связанные с очисткой, такие как удаление слушателей событий и остановка таймеров. После вызова этого метода компонент удаляется из DOM, и вы можете попрощаться с ним.

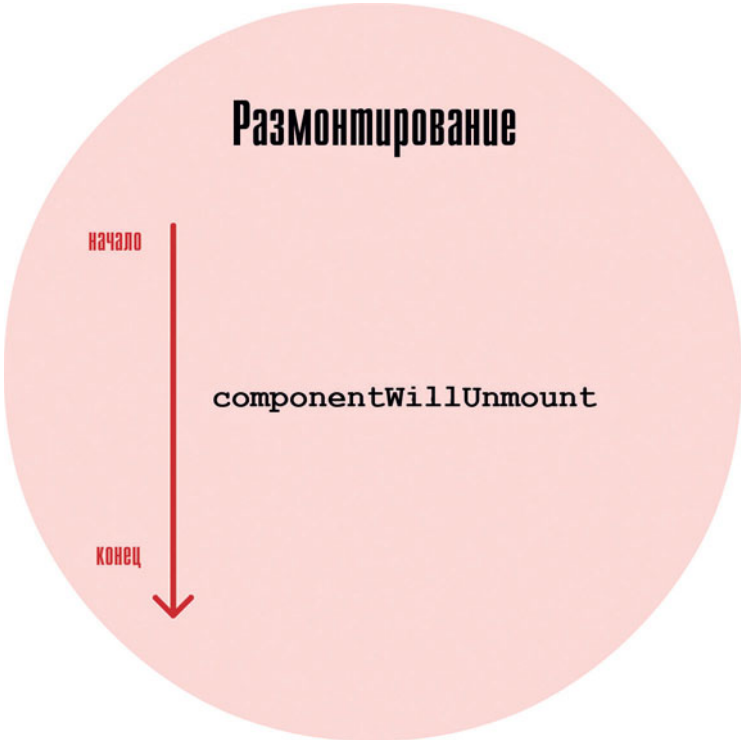


Рис. 11.7. Только один метод жизненного цикла активен, когда компонент уничтожается и удаляется из DOM

Заключение

Работа с компонентами очень увлекательна. Поначалу кажется, что в них мало что происходит. Как в хорошем документальном фильме об океанах, когда мы смотрим все глубже и глубже и видим совершенно другой мир. Оказывается, React постоянно отслеживает и уведомляет компоненты каждый раз, когда происходит что-то интересное. Все это делается с помощью (чрезвычайно скучных) методов жизненного цикла, на которые мы потратили всю эту главу. Теперь вы знаете, чем занимается каждый метод жизненного цикла и когда он вызывается, и все эти знания в один прекрасный день вам пригодятся. Все, что вы узнали, — это не просто тривиальное знание, хотя ваши друзья и будут впечатлены, если вы сможете описать каждый метод жизненного цикла по памяти. Попробуйте удивить их своим новым приложением.

> Примечание: Если у вас возникнут какие-либо проблемы, задайте вопрос!
>
> Если у вас есть какие-либо вопросы или код не работает ожидаемым образом, не стесняйтесь задать вопрос! Опубликуйте свой вопрос на форуме **forum.kirupa.com** и получите помощь от самых дружелюбных и знающих людей в Интернете!

Глава 12
Доступ к элементам DOM в React

Иногда необходимо напрямую обращаться к свойствам и методам HTML-элемента. В среде React, где JSX — эталон идеальной разметки, почему вам нужно иметь дело с ужасным HTML-кодом? Как вы узнаете (если еще не знали), во многих случаях непосредственное обращение к HTML-элементам через API DOM в JavaScript проще, чем возиться с ними с помощью React. Чтобы привести в пример одну из таких ситуаций, создадим палитру, показанную на рис. 12.1.

Рис. 12.1. Приложение-палитра

Если у вас есть доступ к Интернету, вы можете просмотреть приложение по адресу **www.kirupa.com/react/examples/colorizer.htm**.

Палитра окрашивает белый квадрат цветом, который вы укажете. Чтобы увидеть приложение в действии, введите значение цвета в текстовое поле и нажмите кнопку **OK**. (Если вы не знаете, какое значение цвета ввести, введите **yellow**.) После того, как вы укажете цвет и подтвердите действие, белый квадрат окрасится в указанный вами цвет (см. рис. 12.2).

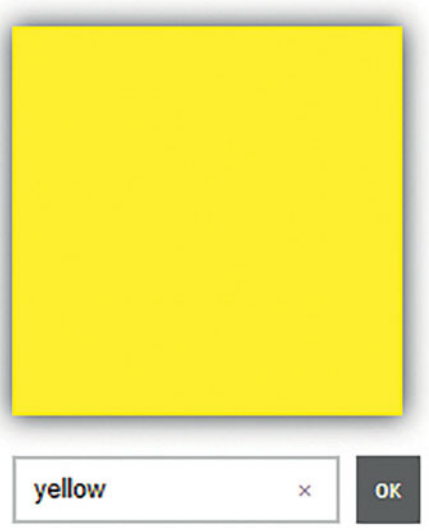


Рис. 12.2. Белый цвет превратился в желтый

Тот факт, что квадрат меняет цвет на любой другой с допустимым значением цвета, переданным вами, довольно удивителен, но это не то, на чем вы должны сосредоточиться. Вместо этого обратите внимание на текстовое поле и кнопку **OK**. Обратите внимание, что кнопка получает фокус, а указанное значение цвета остается в поле. Если вы хотите ввести другое значение цвета, вам нужно вернуть фокус в текстовое поле и удалить указанное там значение. Это опциональная функция, и мы можем улучшить приложение с точки зрения удобства использования.

Было бы круто, если бы мы могли сбросить введенное значение цвета и вернуть фокус в текстовое поле сразу после нажатия кнопки **OK**. Это означало бы, что если бы указали значение фиолетового цвета и нажали бы кнопку **OK**, то в результате увидели бы что-то похожее на рис. 12.3.

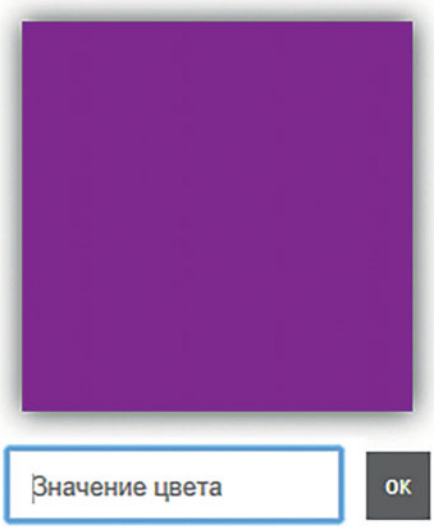


Рис. 12.3. Отобразился фиолетовый цвет, а текстовое поле готово для ввода следующего значения цвета

Введенное значение фиолетового цвета сброшено, а фокус вернулся в текстовое поле. Так можно вводить значения других цветов и легко отправлять их без необходимости перескакивать между текстовым полем и кнопкой. Разве это не лучше?

Правильно реализовать такое поведение с использованием JSX и традиционных методов React сложно. Я даже не буду объяснять, как это сделать. С другой стороны, реализовать такое поведение с помощью API DOM JavaScript для различных HTML-элементов довольно просто. Угадайте, что мы будем делать? В следующих разделах мы воспользуемся так называемыми ссылками (`refs`), доступными в React, чтобы получить доступ к API DOM для HTML-элементов. Мы также рассмотрим порталы, которые позволят нам рендерить контент в любой HTML-элемент на странице.

Код приложения «Палитра»

Чтобы разобраться со ссылками и порталами, мы изменим код приложения «Палитра», который вы видели ранее. Его код выглядит следующим образом:

```
<! DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <title>Палитра</title>
  <script src="https://unpkg.com/react@16/umd/react.
              development.js"></script>
  <script src="https://unpkg.com/react-dom@16/umd/react-dom.
              development.js"></script>
  <script src="https://unpkg.com/babel-standalone@6.15.0/
              babel.min.js"></script>

  <style>
    #container {
      padding: 50px;
      background-color: #FFF;
    }
    .colorSquare {
      box-shadow: 0px 0px 25px 0px #333;
      width: 242px;
      height: 242px;
      margin-bottom: 15px;
    }
    .colorArea input {
      padding: 10px;
      font-size: 16px;
      border: 2px solid #CCC;
    }
    .colorArea button {
      padding: 10px;
```

```
      font-size: 16px;
      margin: 10px;
      background-color: #666;
      color: #FFF;
      border: 2px solid #666;
    }
    .colorArea button: hover {
      background-color: #111;
      border-color: #111;
      cursor: pointer;
    }
  </style>
</head>

<body>
  <div id="container"></div>
  <script type="text/babel">

    class Colorizer extends React.Component {
      constructor(props) {
        super(props);

        this.state = {
          color: "",
          bgColor: "white"
        };

        this.colorValue = this.colorValue.bind(this);
        this.setNewColor = this.setNewColor.bind(this);
      }

      colorValue(e) {
        this.setState({
          color: e.target.value
        });
```

```
    }

    setNewColor(e) {
      this.setState({
        bgColor: this.state.color
      });

      e.preventDefault();
    }

    render() {
      var squareStyle = {
        backgroundColor: this.state.bgColor
      };

    return (
      <div className="colorArea">
        <div style={squareStyle} className="colorSquare"></div>

        <form onSubmit={this.setNewColor}>
          <input onChange={this.colorValue}
          placeholder="Значение цвета"></input>
          <button type="submit">ok</button>
        </form>
      </div>
    );
  }
}

ReactDOM.render(
  <div>
    <Colorizer />
  </div>,
  document.querySelector("#container")
);
```

```
  </script>
</body>

</html>
```

Остановитесь на несколько минут, посмотрите код и поймите, насколько он соответствует нашему примеру. Вы не должны увидеть ничего нового. Если вы хорошо понимаете этот код, пришло время узнать о ссылках.

Знакомство со ссылками

Как вы уже знаете, в различных методах визуализации мы пишем HTML-подобный JSX-код. JSX-код — это описание того, как должен выглядеть DOM. Он не представляет собой фактический HTML-код, несмотря на то, что он очень похож на него. Чтобы обеспечить связь между JSX-кодом и конечными HTML-элементами в DOM, React предоставляет нам **ссылки** (`refs`).

Способ работы ссылок немного странный. Самый простой способ понять его — использовать ссылки. Взгляните на метод `render` в коде палитры:

```
render() {
  var squareStyle = {
    backgroundColor: this.state.bgColor
  };

  return (
    <div className="colorArea">
      <div style={squareStyle} className="colorSquare"></div>

      <form onSubmit={this.setNewColor}>
        <input onChange={this.colorValue} placeholder="Значение
                цвета"></input>
        <button type="submit">OK</button>
```

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

Внутри этого метода `render` мы возвращаем большой фрагмент JSX, представляющий (среди прочего) элемент `input`, которому мы передаем значение цвета. Мы хотим получить доступ к элементам ввода в представлении DOM, чтобы мы могли вызывать некоторые API с помощью JavaScript.

Способ применения ссылок заключается в установке атрибута `ref` элементу, на который мы хотим сослаться:

```
  render() {
    var squareStyle = {
      backgroundColor: this.state.bgColor
    };

    return (
      <div className="colorArea">
        <div style={squareStyle} className="colorSquare"></div>

        <form onSubmit={this.setNewColor}>
          <input onChange={this.colorValue}
          ref={}
          placeholder="Значение цвета"></input>
          <button type="submit">OK</button>
        </form>
      </div>
    );
  }
```

Поскольку нас интересует элемент ввода, к нему привязан атрибут `ref`. На данном этапе атрибут `ref` пуст. То, что вы обычно устанавливаете как значение атрибута `ref`, является функцией

обратного вызова JavaScript. Эта функция вызывается автоматически, когда монтируется компонент, содержащий данный метод `render`. Если мы установим значение атрибута `ref` равное простой функции JavaScript, которая хранит ссылку на ссылочный элемент DOM, код будет выглядеть примерно так:

```
render() {
  var squareStyle = {
    backgroundColor: this.state.bgColor
  };

  var self = this;

  return (
    <div className="colorArea">
      <div style={squareStyle} className="colorSquare"></div>

      <form onSubmit={this.setNewColor}>
        <input onChange={this.colorValue}
        ref={
          function(el) {
          self._input = el;
          }
        }
        placeholder="Значение цвета"></input>
        <button type="submit">OK</button>
      </form>
    </div>
  );
}
```

Результат работы этого кода, который выполняется после того, как смонтированы компоненты, прост: мы можем получить доступ к HTML, представляющему элемент `input`, из любой позиции нашего компонента, используя метод `self.input`. Уделите пару минут,

чтобы разобраться, как функционируют выделенные строки кода. Когда вы закончите, мы вместе пройдемся по этому коду.

Во-первых, функция обратного вызова выглядит следующим образом:

```
function(el) {
  self._input = el;
}
```

Эта анонимная функция вызывается при монтировании компонента, а ссылка на конечный HTML-элемент DOM передается в качестве аргумента. Мы перехватываем этот аргумент с помощью идентификатора `el`, но вы можете использовать любое имя для этого аргумента. Тело функции обратного вызова устанавливает пользовательское свойство, `_input`, как значение элемента DOM. Чтобы гарантировать, что мы создадим это свойство на нашем компоненте, мы используем переменную `self` для создания замыкания — `this` относится к нашему компоненту, а не к самой функции обратного вызова (любая функция, которая передается как аргумент).

Давайте сосредоточимся на том, что мы можем сделать теперь, когда у нас есть доступ к элементу `input`. Наша цель — очистить содержимое элемента ввода и сфокусироваться на нем после отправки формы. Код будет работать в методе `setNewColor`, для чего внесите следующие изменения (выделены цветом):

```
setNewColor(e) {
  this.setState({
    bgColor: this.state.color
  });

  this._input.focus();
  this._input.value = "";

  e.preventDefault();
}
```

Вызов метода `this._input.value = ""` очищает введенное значение. Мы фокусируемся на элементе ввода, вызвав функцию `this._input.focus()`. Вся наша работа заключалась в том, чтобы добавить эти две строки; нам нужен был способ, чтобы метод `this._input` указывал на HTML-элемент, представляющий элемент `input`, который мы определили в JSX-коде. Затем мы можем вызвать свойство `value` и метод `focus`, которые API DOM предоставляет для этого элемента.

Примечание. Упрощение с помощью стрелочных функций ES6

Изучать React достаточно сложно, поэтому я пытался избежать принуждения к использованию методов ES6. Когда дело доходит до работы с атрибутом `ref`, использование стрелочных функций для работы с функцией обратного вызова упрощает дело. Это один из тех случаев, для которых я рекомендую использовать технику ES6.

Как вы видели, чтобы присвоить свойство компонента ссылочному HTML-элементу, мы сделали что-то вроде этого:

```
<input
  ref={
    function(el) {
      self._input = el;
    }
  }>
</input>
```

Для взаимосвязи мы создали инициализированную переменную `self`, чтобы убедиться, что мы создали свойство `_input` компонента. Это кажется неудобным.

Используя стрелочные функции, можно упростить код так:

```
<input
  ref={
    (el) => this._input = el
  }>
</input>
```

> Конечный результат идентичен тому, что мы все время видели. Так как стрелочные функции зависимы от контекста, вы можете использовать их в теле функции и ссылаться на компонент без лишних хлопот. Нет необходимости во внешнем эквиваленте переменной `self`!

Использование порталов

Вам нужно знать еще один трюк, связанный с DOM. До сих пор мы работали с HTML только в контексте того, что генерирует JSX-код, либо из одного или нескольких компонентов. Это означает, что мы ограничены иерархией DOM, которую родительские компоненты налагают на нас. Наличие произвольного доступа к любому элементу DOM в любом месте страницы не представляется возможным. Или представляется? Как оказалось, вы можете отображать JSX-контент для любого элемента DOM в любом месте страницы; вы не ограничены отправкой своего JSX-кода в родительский компонент. Причина этого волшебства — функционал, известный как **порталы**.

То, как мы используем портал, очень похоже на то, что мы делаем с методом `ReactDOM.render`. Мы указываем JSX-код, который хотим визуализировать, и указываем элемент DOM, который мы тоже хотим визуализировать.

Чтобы увидеть все это в действии, вернитесь к нашему примеру и добавьте следующий сестринский элемент `h1` чуть выше определения элемента `div` с идентификатором `container`:

```
<body>

  <h1 id="colorHeading">Палитра</h1>

  <div id="container"></div>
        .
        .
        .
```

Затем добавьте следующее правило стиля в элемент `style`, чтобы наш элемент `h1` стал более привлекательным:

```
#colorHeading {
  padding: 0;
  margin: 50px;
  margin-bottom: -20px;
  font-family: sans-serif;
}
```

После добавления правил стилей давайте просмотрим на наше приложение, чтобы убедиться, что HTML и CSS-код, который мы добавили, приводят к желанному результату (рис. 12.4):

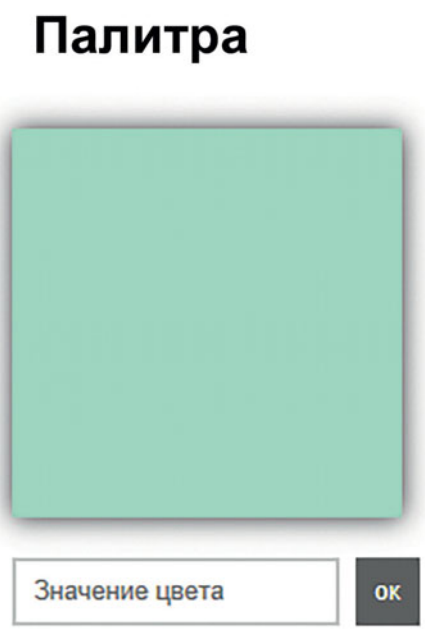


Рис. 12.4. Так выглядит пример прямо сейчас!

Вот что нам нужно. Мы хотим изменить значение элемента `h1`, чтобы отобразить имя цвета, который отображается в данный

момент. Следует подчеркнуть, что элемент `h1` является родственником элемента `div container`, где наше приложение настроено на визуализацию.

Чтобы выполнить необходимые действия, вернитесь к методу `render` компонента `Colorizer` и добавьте следующую выделенную строку:

```
return (
  <div className="colorArea">
    <div style={squareStyle} className="colorSquare"></div>

    <form onSubmit={this.setNewColor}>
      <input onChange={this.colorValue}
      ref={
        function(el) {
          self._input = el;
        }
      }
      placeholder="Значение цвета"></input>
      <button type="submit">go</button>
    </form>
    <ColorLabel color={this.state.bgColor}/>
  </div>
);
```

Мы создаем компонент `ColorLabel` и объявляем свойство с именем `color` с его значением, заданным как свойство `bgColor state`. Мы еще не создали этот компонент, поэтому добавьте следующие строки чуть выше вызова метода `ReactDOM.render`:

```
var heading = document.querySelector("#colorHeading");

class ColorLabel extends React.Component {
  render() {
    return ReactDOM.createPortal(
```

```
        ": " + this.props.color,
        heading
      );
    }
  }
```

Мы ссылаемся на элемент `h1` с помощью переменной `heading`. Это старый трюк. Для нового взгляните на метод `render` компонента `ColorLabel`. В частности, обратите внимание, как выглядит инструкция `return`. Мы возвращаем результат вызова метода `ReactDOM.createPortal()`:

```
  class ColorLabel extends React.Component {
    render() {
      return ReactDOM.createPortal(
        ": " + this.props.color,
        heading
      );
    }
  }
```

Метод `ReactDOM.createPortal()` принимает два аргумента: JSX для вывода и элемент DOM для вывода данного JSX-контента. JSX-контент, который мы выводим, — это всего лишь некоторые символы форматирования и значение цвета, которое мы передали в качестве свойства:

```
  class ColorLabel extends React.Component {
    render() {
      return ReactDOM.createPortal(
        ": " + this.props.color,
        heading
      );
    }
  }
```

Элемент DOM, который мы выводим, — это элемент `h1`, на который ссылается переменная `heading`:

```
class ColorLabel extends React.Component {
  render() {
    return ReactDOM.createPortal(
      ": " + this.props.color,
      heading
    );
  }
}
```

Когда вы запустите приложение и измените цвет, обратите внимание на то, что произойдет. Цвет, указанный в элементе ввода, отображается в заголовке (рис. 12.5):

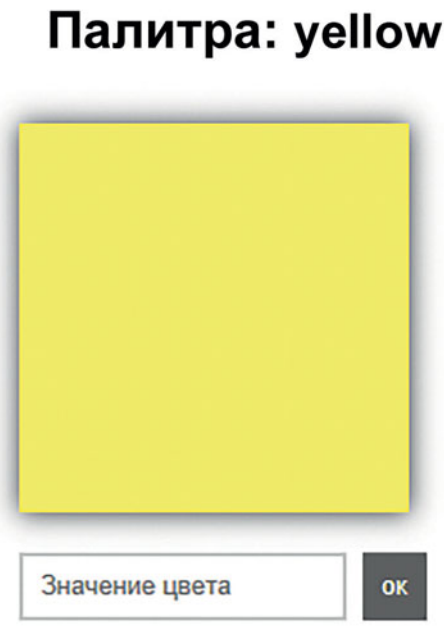


Рис. 12.5. Теперь заголовок содержит значение цвета

Не помешает еще раз повторить, что работа с элементом `h1` выходит за рамки нашего основного приложения React, который выводится на элемент `container div`. Используя порталы, мы имеем прямой доступ к любому элементу DOM страницы и можем отображать в нем контент, минуя традиционную иерархию «предок/потомок», которой пользовались до сих пор.

Заключение

В большинстве случаев все, что вы хотите сделать, будет в рамках JSX-кода, который вы напишете. Иногда, однако, нужно выйти за рамки React. Несмотря на то, что все, что мы делаем, это визуализируем HTML-документ, наше приложение React похоже на самодостаточный тропический остров в документе; вы никогда не увидите фактический HTML-код, который скрыт под песком. Чтобы раскопать HTML-код внутри острова и связаться с элементами, находящимися за пределами острова, мы рассмотрели две функции: ссылки и порталы. Ссылки позволяют получить доступ к базовому элементу HTML за пределами JSX. Порталы предоставляют ваше содержимое любому элементу в DOM, к которому у вас есть доступ. С этими двумя подходами вы сможете легко решить любую проблему, с которой вам придется иметь дело в DOM.

> Примечание: Если у вас возникнут какие-либо проблемы, задайте вопрос!
>
> Если у вас есть какие-либо вопросы или код не работает ожидаемым образом, не стесняйтесь задать вопрос! Опубликуйте свой вопрос на форуме **forum.kirupa.com** и получите помощь от самых дружелюбных и знающих людей в Интернете!

Глава 13
Настройка среды разработки React

Последняя важная тема, о которой мы поговорим, — это настройка среды разработки для создания приложений React. До сих пор мы собирали приложения React, включая несколько файлов сценариев:

```
<script src="https://unpkg.com/react@16/umd/react.
development.js"></script>
<script src="https://unpkg.com/react-dom@16/umd/react-dom.
development.js"></script>
<script src="https://unpkg.com/babel-standalone@6.15.0/
babel.min.js"></script>
```

Эти файлы сценариев загружали не только библиотеки React, но и компилятор Babel, помогающий браузеру обрабатывать такие странные вещи, как JSX (рис. 13.1):

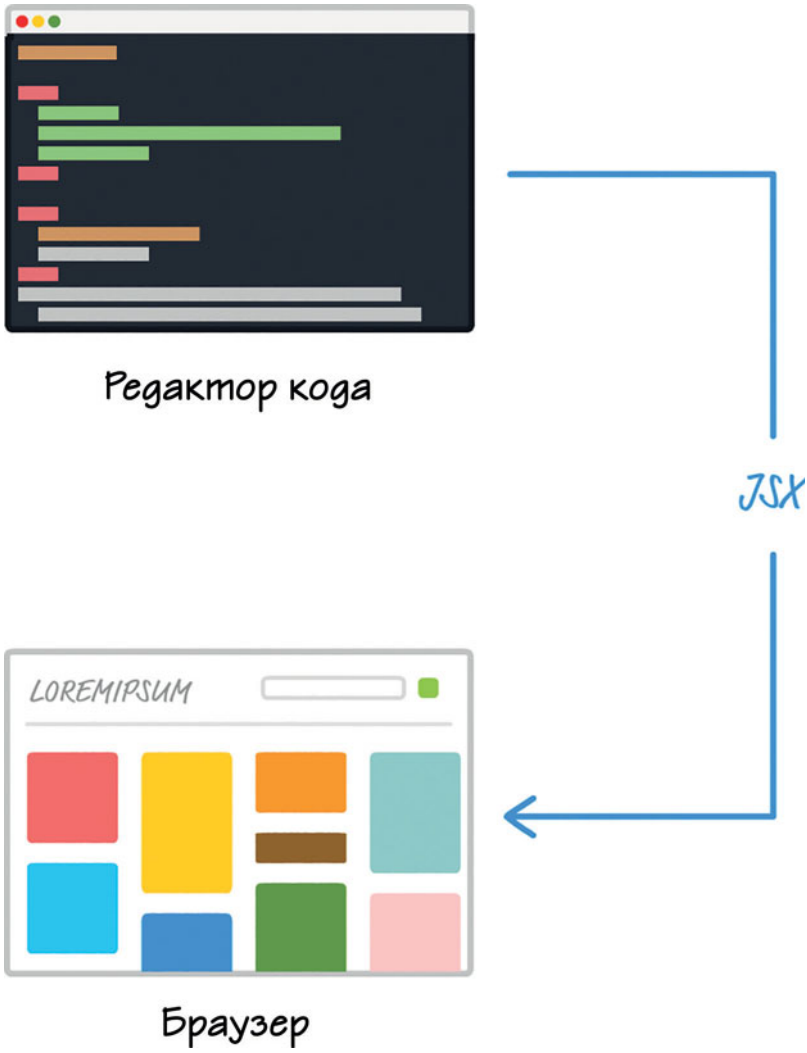


Рис. 13.1. Схема работы встроенного транслятора JSX

Как упоминалось ранее, в этом подходе недостатком является производительность. Поскольку браузер обрабатывает все загружаемые страницы, как это обычно и бывает, он также несет ответственность за преобразование JSX-кода в JavaScript-кода. Это преобразование — трудоемкий процесс, который хорош во время разработки, но не подходит, если каждый пользователь приложения должен отплатить за это снижением производительности.

Решение состоит в том, чтобы настроить среду разработки так, чтобы преобразование JSX — JS обрабатывалось как часть создания приложения (рис. 13.2):

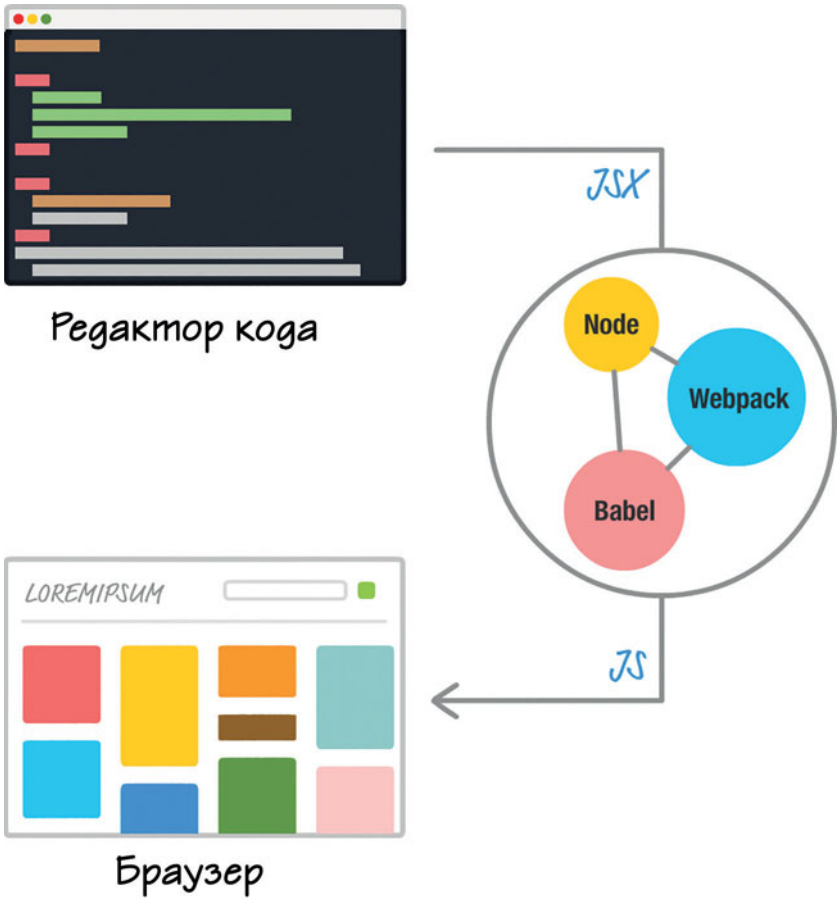


Рис. 13.2. Так выглядит правильная настройка среды разработки с JSX!

Теперь браузер загружает приложение и обращается к преобразованному (и оптимизированному) файлу JavaScript. Хорошее решение, не так ли? Единственная причина, по которой мы не говорили об этом до сих пор, — это **простота**. Изучать React достаточно сложно. Добавление описаний инструментов сборки и настройки среды разработки в процессе обучения React повлияло бы на сложность

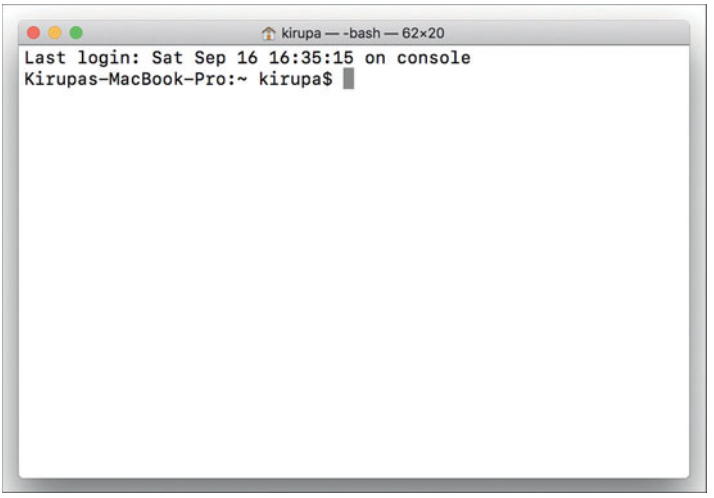
материала. Теперь, когда у вас есть полное понимание принципов работы React, пришло время настроек.

В следующих разделах мы рассмотрим один из способов настройки среды разработки с использованием комбинации Node js, Babel и webpack. Если все это звучит странно, не беспокойтесь. Мы будем использовать действительно отличное решение, созданное Facebook, упрощающее процесс до предела.

Проект Create React

Несколько лет назад подготовка среды разработки было огромной головной болью, поскольку процесс включал ручную настройку всех инструментов, о которых мы говорили. Вам бы пришлось просить о помощи своего друга — реально продвинутого программиста. Возможно, вы даже начали бы сомневаться, стоит ли учиться программированию React. К счастью, был создан проект Create React (**github.com/facebookincubator/create-react-app**), который значительно упростил процесс настройки среды разработки React. Вы запускаете несколько команд в оболочке командной строки, и без вашего участия создается проект React со всеми необходимыми конфигурациями.

Прежде чем начать работу, убедитесь, что установлена последняя версия Node js (**nodejs.org**). Затем откройте свою любимую оболочку командной строки. Если вы не слишком хорошо знакомы с командами, не беспокойтесь. В Windows запустите либо командную строку, либо оболочку BASH. В macOS запустите приложение Терминал (Terminal). Вы увидите следующее:



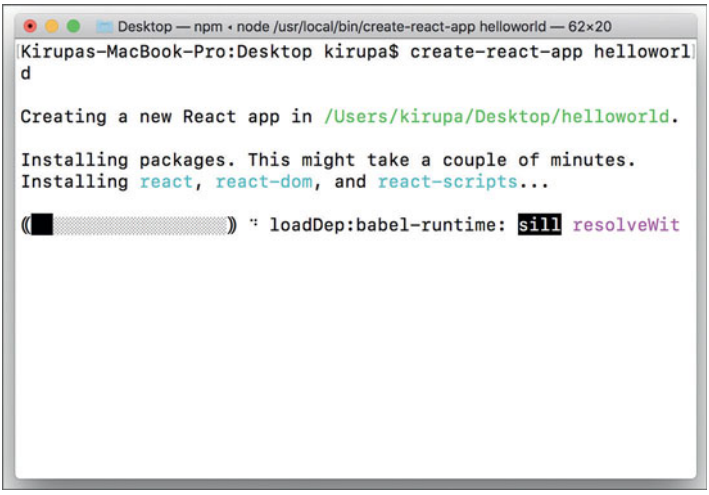
Это оболочка командной строки — причудливое окно с мигающим курсором, который позволяет вводить в него команды. Первое, что вам нужно сделать, это создать проект Create React. Введите следующую команду в командной строке и нажмите клавишу **Enter**/**Return**:

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

Процесс может занять от нескольких секунд до нескольких минут, но как только установка завершится, можно создать новый проект React. Перейдите в папку, в которой вы хотите создать свой новый проект, — это может быть рабочий стол, каталог с документами и т. д. Когда вы перешли к папке в командной строке, введите следующую команду, чтобы создать новый проект в выбранной папке:

```
create-react-app helloworld
```

Вы увидите следующее:



Когда команда будет полностью выполнена, у вас появится проект **helloworld**, созданный за вас. Не беспокойтесь о том, что произошло за кадром; мы рассмотрим содержимое проекта чуть позже. Пока что первое, что нужно сделать, это проверить созданный проект. Перейдите в папку *helloworld* созданного проекта, введя следующую команду:

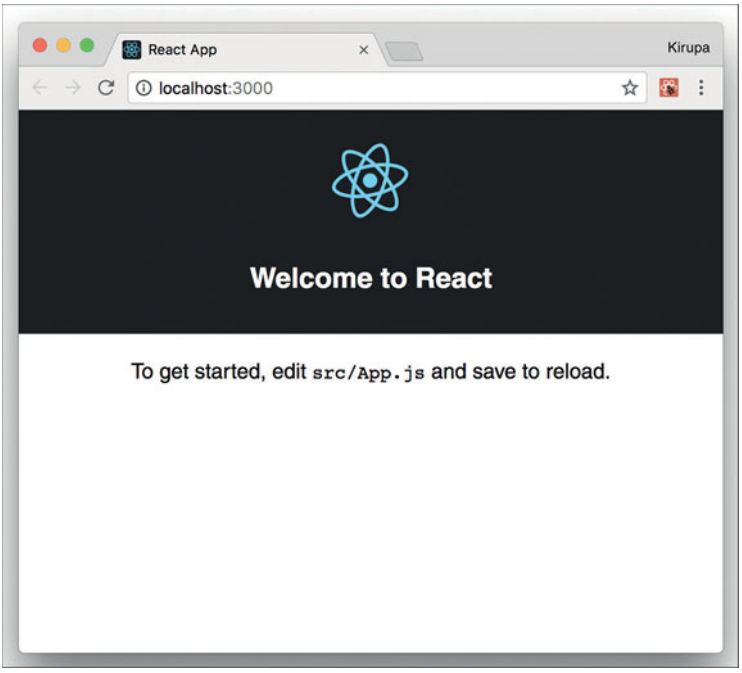
```
cd helloworld
```

В этой папке введите следующую команду, чтобы проверить приложение:

```
npm start
```

Если у вас установлен менеджер `yarn`, Create предпочтет использовать его вместо `npm` для установки, и вы увидите экранные инструкции, в которых говорится, что вместо команды `npm start` следует выполнять команду `yarn start`.

Проект будет собран, будет запущен локальный веб-сервер, и вы увидите запущенный проект, как показано на следующем рисунке:



Если процесс завершился успешно, вы должны увидеть то же самое. Если вы впервые создаете новый проект React с помощью командной строки, поздравляем! Это действительно большой шаг. Однако мы не закончили. Теперь нам нужно сделать несколько шагов назад и пересмотреть то, что именно произошло.

Анализ произошедшего

На данном этапе мы видим, что некое дефолтное содержимое создано с помощью команды `create-react-app`. Это не очень понятно. Во-первых, давайте посмотрим, что именно было создано. Структура папок и файлов после выполнения команды `create-react-app helloworld` будет выглядеть так, как показано на рис. 13.3:

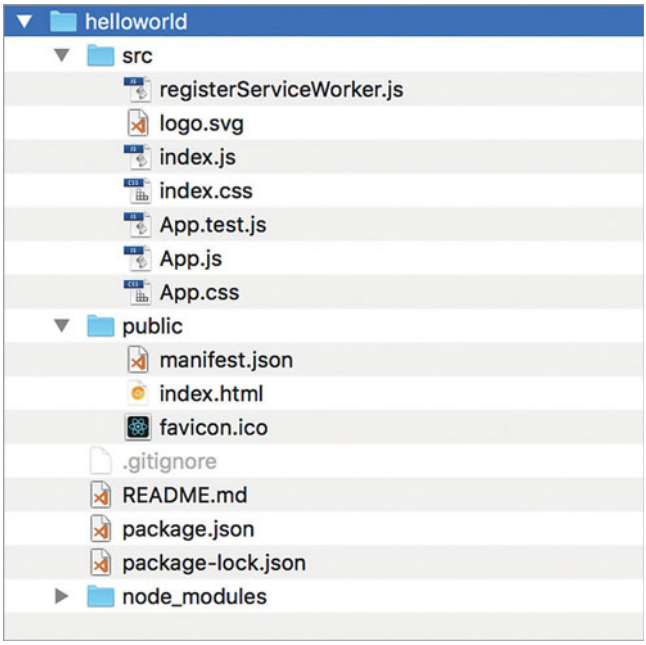


Рис. 13.3. Структура файлов и папок проекта

Файл Index.html в папке *public* загружается в браузер. Если вы откроете этот файл в редакторе, вы легко поймете его код. Вот содержимое этого файла без комментариев (я их удалил):

```
<! DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width,
              initial-scale=1, shrink-to-fit=no">
    <meta name="theme-color" content="#000000">

    <link rel="manifest" href="%PUBLIC_URL%/manifest.json">
    <link rel="shortcut icon" href="%PUBLIC_URL%/favicon.ico">

    <title>React App</title>
```

```
  </head>
  <body>
    <noscript>
      You need to enable JavaScript to run this app.
    </noscript>

    <div id="root"></div>

  </body>
</html>
```

На что нужно обратить внимание, — это элемент `div` с идентификатором `root`. Здесь будет выводиться контент приложения React. Уточню, содержимое приложения React со всем JSX-кодом хранится в папке *src*. Начальный файл приложения React носит имя *index.js*:

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

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

Обратите внимание на вызов `ReactDOM.render`, который ищет элемент `root`, вызываемый из файла *index.html*. Вы также увидите группу инструкций `import` в верхней части кода страницы. Эти инструкции импорта являются частью JavaScript-кода, называемого **модулями**. Цель модулей — разделить функциональность приложения на более мелкие части. Когда приходит время использовать некий фрагмент кода, вы импортируете только необходимый модуль, а не все сразу. Некоторые из модулей, которые вы импортируете, относятся к коду проекта. Другие, такие как `React` и `ReactDOM`, находятся вне проекта, но также могут быть импортированы. Я еще

многое могу рассказать о загрузке модулей, но этого достаточно. Давайте оставим эту тему.

В коде мы импортируем библиотеки `React` и `React-DOM`. Это должно напомнить вам, как мы добавляли элементы `script` для них ранее. Мы также импортируем файл CSS, служебный сценарий, на который мы будем ссылаться по имени `registerServiceWorker`, и компонент React с именем `App`.

Компонент `App` — это наша следующая остановка, поэтому, чтобы увидеть его код, откройте файл *App.js*:

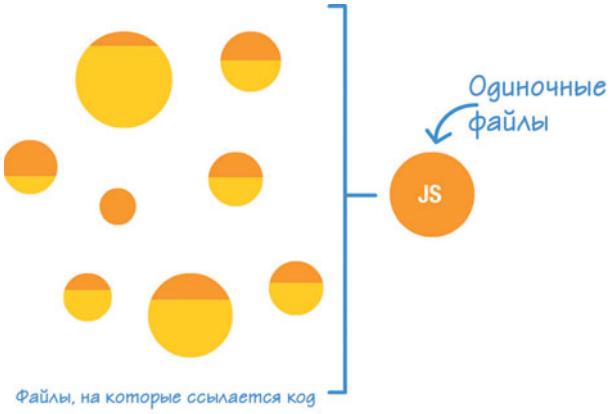
```
import React, {Component} from 'react';
import logo from './logo.svg';
import './App.css';

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

export default App;
```

Обратите внимание, что файл *App.js* содержит собственные инструкции `import`. Некоторые из них, например импортирующие

библиотеки `React` и `Component`, необходимы, учитывая предназначение кода. Здесь интересна последняя строка: `export default app`. Она содержит команду `export` и имя, которое наш проект будет использовать для идентификации экспортируемого модуля. Вы будете использовать это имя при импорте модуля `App` в других частях проекта, например, в файле *index.js*. Кроме того, также импортируются изображения и файлы CSS, необходимые для корректного отображения этой страницы.

Теперь вы видели другой способ структурирования кода с применением некоторых новых ключевых слов. Какова цель всего этого? Эти модули, инструкции `import` и `export` — это лишь тонкости, чтобы код приложения стал более управляемым. Вместо того чтобы определять все в одном гигантском файле, вы можете разбить свой код и связанные с ним ресурсы на несколько файлов. В зависимости от того, на какие файлы вы ссылаетесь и какие файлы загружаются перед другими файлами, таинственный процесс сборки (в настоящее время начинающийся с команды `npm start`) может оптимизировать конечный результат различными способами, о которых нам не нужно беспокоиться.

Важно отметить: то, что вы делаете с кодом, не влияет на функциональность финального приложения. За кадром, когда мы готовы протестировать приложение, происходит этап сборки. Этот этап сборки придает смысл хаосу различных файлов и компонентов, которые вы импортируете, чтобы представить их как легко обрабатываемый набор комбинированных файлов для браузера. Мы получим один JS-файл со всеми соответствующими частями, которые рассмотрели ранее:



Мы также получим один комбинированный CSS-файл. В зависимости от того, что еще вы могли бы настроить, вы могли бы получить другие комбинированные HTML-файлы и многое другое. Все они будут в виде, поддерживаемом браузером. У браузера не будет никакой дополнительной работы. Все будет представлено как обычный код на языках HTML, CSS и JavaScript.

Создание демонстрационного приложения

Теперь, когда вы получили представление о том, как устроен этот проект, давайте изменим пример. Нам нужно отобразить на экране слова «Привет, мир!». Мы рассмотрим процесс, создав компонент `HelloWorld`. В новинку для вас будет не вывод текста на экран; вы уже профи в этом. Внимание следует обратить на структурирование файлов в проекте, чтобы обеспечить *правильное* создание приложения.

Для начала перейдите в каталог *src* и удалите все файлы, которые там есть. Затем создайте файл *index.js*. В этот файл добавьте следующий код:

```
import React from "react";
import ReactDOM from "react-dom";
import HelloWorld from "./HelloWorld";

ReactDOM.render(
  <HelloWorld/>,
  document.getElementById("root")
);
```

Мы импортируем модули `React` и `ReactDOM`. Мы также импортируем компонент `HelloWorld`, который указан в вызове `ReactDOM.render`. Этого компонента не существует, поэтому мы решим проблему позднее.

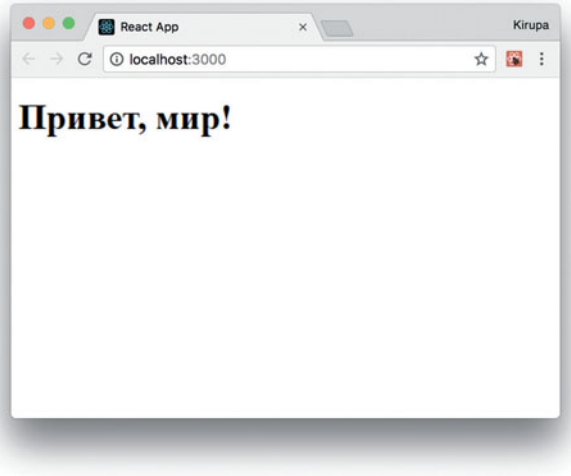
В том же каталоге *src*, в котором мы сейчас находимся, создайте файл *HelloWorld.js*. Затем откройте его и измените, добавив следующий код:

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

class HelloWorld extends Component {
  render() {
    return (
      <div className="helloContainer">
        <h1>Привет, мир!</h1>
      </div>
    );
  }
}

export default HelloWorld;
```

Найдите минутку, чтобы проанализировать добавленный код. Вы не должны здесь видеть ничего интересного — скучная инструкция `import`, компонент `HelloWorld`, который выводит текст на экран, и (в последней строке) код, который экспортирует компонент `HelloWorld`, чтобы он мог импортироваться другим модулем, таким как *index.js*.

С этими изменениями мы можем протестировать приложение. Убедитесь, что вы сохранили все изменения. Вернитесь в оболочку командной строки и выполните команду `npm start`. Если приложение уже работает, вы увидите автоматически обновленную страницу с последними изменениями. Если этого не произошло или приложение завершило работу, нажмите сочетание клавиш **Ctrl+C**, чтобы остановить сеанс, и снова введите команду `npm start`.

Вы должны увидеть следующее:



Если это так, мои поздравления! Приложение работает правильно, хотя и выглядит слишком простым. Давайте исправим это, добавив каскадные таблицы стилей. Создайте таблицу стилей с именем *index.css*, и добавьте в файл следующие правила стилей:

```
body {
  display: flex;
  align-items: center;
  justify-content: center;
  min-height: 100vh;
  margin: 0;
}
```

При создании приложений добавление таблицы стилей — только часть всего, что нужно сделать. Также нужно, чтобы вы сослались на созданный документ *index.css* в файле *index.js*. Откройте файл *index.js* и добавьте в него выделенную в листинге инструкцию `import`:

```
import React from "react";
import ReactDOM from "react-dom";
```

```
import HelloWorld from "./HelloWorld";
import "./index.css";

ReactDOM.render(
  <HelloWorld/>,
  document.getElementById("root")
);
```

Если вы вернетесь в браузер, вы заметите, что страница автоматически обновилась со всеми последними изменениями. Вы увидите слова «Привет, мир!», расположенные в центре по вертикали и горизонтали. Неплохо, но можно сделать и красивее.

Последнее, что мы сделаем, — оформим текст более стильно. Мы могли бы добавить соответствующие правила стиля в файл *index.css*, но более подходящее решение — создание нового файла CSS, в котором мы сошлемся только на компонент `HelloWorld`. Конечный результат обоих способов одинаков, но вы же хотите попрактиковаться в группировке связанных файлов (и их зависимостей), чтобы стать более профессиональным разработчиком?

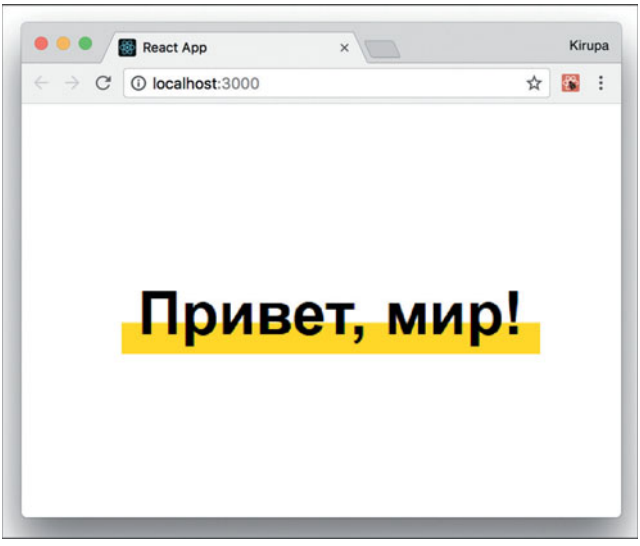
Создайте файл *HelloWorld.css* в папке *src*. Добавьте в него следующие правила стилей:

```
h1 {
    font-family: sans-serif;
    font-size: 56px;
    padding: 5px;
    padding-left: 15px;
    padding-right: 15px;
    margin: 0;
    background: linear-gradient(to bottom,
                white 0%,
                white 62%,
                gold 62%,
                gold 100%);
}
```

Осталось только сослаться на эту таблицу стилей в файле *HelloWorld.js*, поэтому откройте его и добавьте выделенную цветом инструкцию `import`:

```
import React, {Component} from "react";
import "./HelloWorld.css";

class HelloWorld extends Component {
  render() {
    return (
      <div className="helloContainer">
        <h1>Привет, мир!</h1>
      </div>
    );
  }
}

export default HelloWorld;
```

Если вы вернетесь в свой браузер, то в случае успеха увидите следующее:



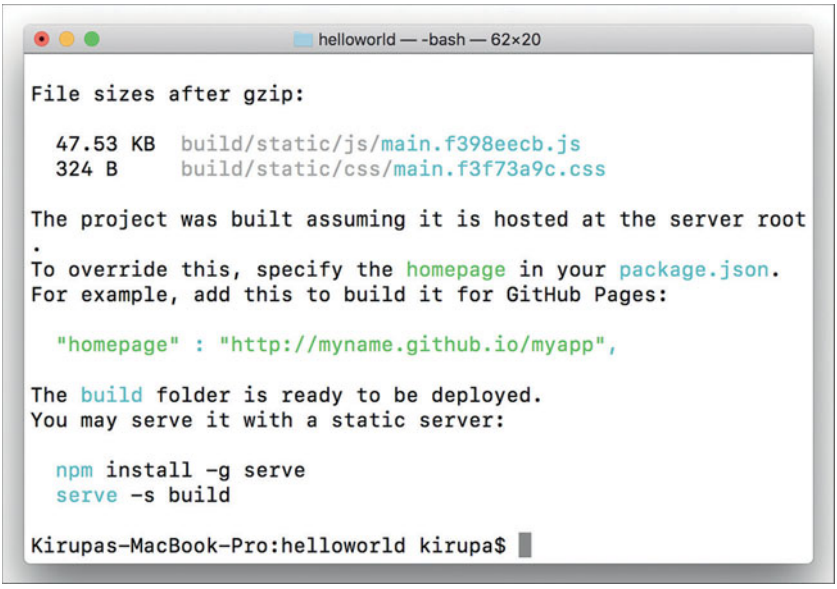
Вы увидите слова «Привет, мир!», но отформатированные круче, чем пару минут назад.

Сборка проекта

Мы почти закончили. Осталось еще кое-что. До сих пор мы создавали это приложение в **режиме разработчика**. В этом режиме код не минимизируется, а некоторые из инструкций выполняются медленно/тщательно, чтобы было легче решать проблемы. Теперь пришло время отправить приложение потенциальным пользователям, и нам нужно самое быстрое и компактное решение. Для этого мы можем вернуться в оболочку командной строки и ввести следующую команду (перед этим нажав сочетание клавиш **Ctrl+C**):

```
npm run build
```

Выполнение команды займет несколько минут, пока создается набор оптимизированных файлов. Как только она завершится, вы увидите текст подтверждения:



Когда процесс завершится, вы можете следовать экранным инструкциям, чтобы развернуть приложение на своем сервере или протестировать его локально, используя популярный пакет `serve`.

Также найдите пару минут, чтобы просмотреть все сгенерированные файлы. Вы увидите только HTML, CSS и JS-файлы. Нет JSX. Нет множества JS-файлов. Есть только один JS-файл, содержащий всю логику, согласно которой должно работать приложение.

Заключение

В этой главе мы использовали удивительное решение `Create React` для создания приложения React современным способом. Если вы использовали его впервые, то вам стоит лучше ознакомиться с этим подходом. Мы будем применять команду `create-react-app` в будущих проектах React; ранее описанный «браузерный» подход был описан для того, чтобы помочь вам изучить основы. Под названием `Create React` скрывается большой комплекс настроек, связанный с `Node JS`, `Babel`, `webpack` и другими компонентами. В этом его мощь, а также и самая большая слабость.

Если вы хотите выйти за пределы красной дорожки, прокладываемой `Create React`, вам нужно будет изучить сложности, скрытые под ней. Ответы на все эти вопросы выходят за рамки нашей книги. Для начала взгляните на содержимое различных JS-файлов в каталоге *node_modules/react_ scripts/scripts*.

> Примечание: Если у вас возникнут какие-либо проблемы, задайте вопрос!
>
> Если у вас есть какие-либо вопросы или код не работает ожидаемым образом, не стесняйтесь задать вопрос! Опубликуйте свой вопрос на форуме **forum.kirupa.com** и получите помощь от самых дружелюбных и знающих людей в Интернете!

Глава 14
Работа с внешними данными в React

Работа с внешними данными сегодня довольно обыденная процедура в веб-приложениях. Процесс обычно выглядит следующим образом:

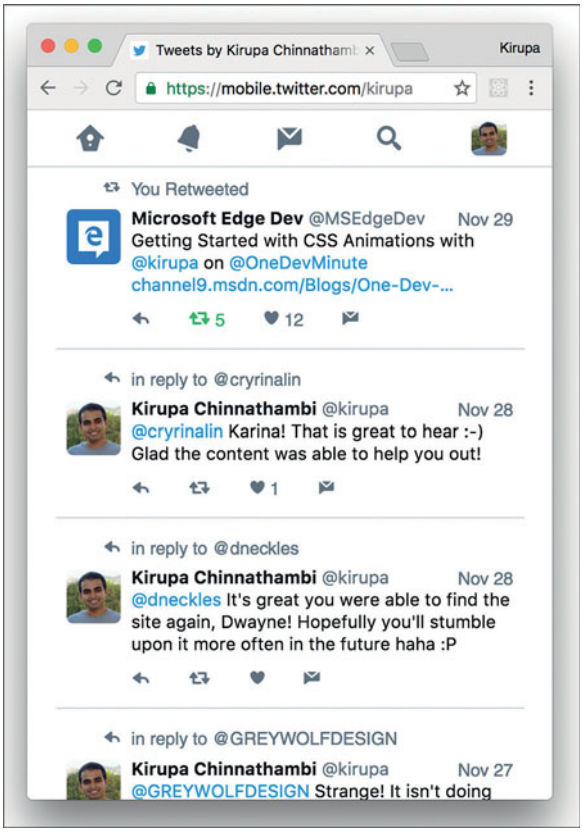
1. Приложение отправляет запрос для получения определенных данных на удаленный сервер.

2. Удаленный сервер получает запрос и отправляет обратно необходимые данные.

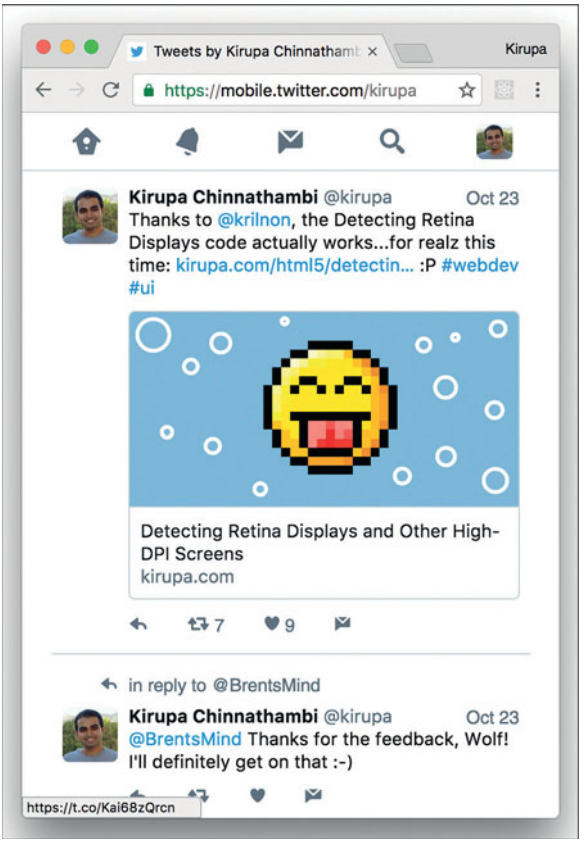
3. Приложение получает данные.

4. Приложение обрабатывает данные и отображает их пользователю.

Почти все популярные сайты следуют этим четырем шагам… Facebook, Amazon, Twitter, Instagram, Gmail, KIRUPA и т. д. Когда вы загружаете страницы на любом из этих сайтов, все они отображают некоторые данные.

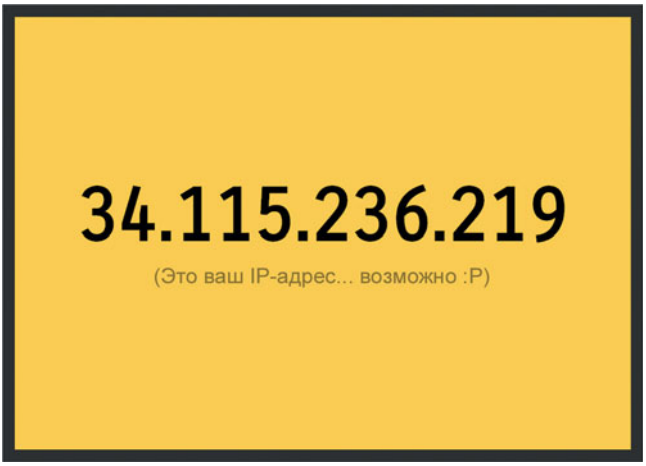


Для сохранения малого размера страницы не все данные загружаются сразу. После того, как страница полностью загружена или вы начали взаимодействовать с ней, страница загружает дополнительные данные с сервера и отображает их.



Все это происходит без необходимости обновления страницы или потери состояния, в котором находится страница. Магия за кадром — это немного JavaScript-кода, который отвечает за четыре шага, приведенные ранее. Из этой главы вы узнаете все о JavaScript-коде, необходимом для полноценной работы приложений React.

К концу главы вы создадите простое приложение React, которое выглядит следующим образом (откройте в браузере страницу **www.kirupa.com/react/examples/ipaddress.htm**):



Здесь вы видите IP-адрес вашего устройства. Это и есть наше приложение. Оно, конечно, совсем не сложное, как приведенные ранее примеры (особенно если вы в восторге от Twitter), но описывает все необходимые детали, позволяющие обрабатывать внешние данные в приложениях React.

Основы веб-запросов

Как вы, наверное, уже знаете, Интернет состоит из множества взаимосвязанных компьютеров, называемых серверами. Когда вы занимаетесь серфингом в Интернете и перемещаетесь между веб-страницами, в реальности вы просите браузер запрашивать информацию с таких серверов. Это выглядит следующим образом: браузер отправляет запрос, ждет ответа сервера на запрос и, как только сервер отвечает, обрабатывает запрос. Все это взаимодействие стало возможным благодаря **протоколу HTTP**.

Протокол HTTP представляет общий язык, который позволяет браузеру и прочим программам и устройствам общаться со всеми серверами, расположенными в Интернете. Запросы, которые браузер отправляет от вашего имени с использованием протокола HTTP, известны как **HTTP-запросы**, и эти запросы выходят за рамки простой загрузки новой страницы во время навигации. В общих чертах

(и намного более захватывающих) и большинстве случаев происходит обновление *существующей* страницы с помощью данных, полученных в результате HTTP-запроса.

Например, у вас может быть страница, на которой вы хотите отобразить некоторую информацию о текущем залогинившемся пользователе. Это информация, которую страница могла не содержать сначала, но это данные, которые браузер будет запрашивать при взаимодействии со страницей. Сервер ответит потоком данных, и произойдет обновление страницы с их использованием. Все это, вероятно, звучит немного абстрактно, поэтому я ненадолго остановлюсь и расскажу о возможном HTTP-запросе и ответе в следующем примере.

Чтобы получить информацию о пользователе, нужно совершить HTTP-запрос:

```
GET /user
Accept: application/json

На этот запрос сервер может вернуть следующий ответ:

200 OK
Content-Type: application/json

{
  "name": "Kirupa",
  "url": "http: https://www.kirupa.com"
}
```

Подобный обмен происходит несколько раз, и все это полностью поддерживается в языке JavaScript. Эта технология асинхронно запрашивать и обрабатывать данные с сервера, не требуя навигации/перезагрузки страницы, имеет название: **Ajax**. Аббревиатура расшифровывается как **асинхронный JavaScript и XML**. Несколько лет назад в кругу веб-разработчиков Ajax было модным словом, которое все использовали, чтобы написать те веб-приложения, которые мы считаем сегодня само собой разумеющимися (Twitter, Facebook, Google

Maps, Gmail и т. д.). Они постоянно извлекают данные при взаимодействии со страницей, не требуя полной ее перезагрузки.

В JavaScript объект, который отвечает за отправку и получение HTTP-запросов, называется `XMLHttpRequest`. Этот объект позволяет выполнять несколько операций, важных для создания веб-запросов:

1. Отправка запроса на сервер.
2. Проверка статуса запроса.
3. Получение и анализ ответа на запрос.
4. Определение события `readystatechange`, которое помогает отреагировать на статус запроса.

XMLHttpRequest делает еще кое-что, но пока это неважно.

> Примечание. Почему бы не использовать сторонние библиотеки?
> Существует много сторонних библиотек, упрощающих работу с объектом `XMLHttpRequest`. Используйте их, если хотите, но и непосредственная работа с объектом `XMLHttpRequest` не слишком сложна. Это всего лишь несколько строк кода, причем одних из самых простых (по сравнению со всем, что вы изучали в React).

Время работать с React

Теперь, когда вы достаточно хорошо понимаете, как работают HTTP-запросы и объект `XMLHttpRequest`, пришло время переключить свое внимание на React. Однако я должен предупредить вас, что React мало пригоден для работы с внешними данными. React в первую очередь сфокусирован на представлении (помните, V в модели MVC?). Мы будем писать обычный JavaScript-код внутри *компонента React,*

основной целью которого является обработка веб-запросов, которые мы будем создавать. Мы немного поговорим об элементах дизайна, но давайте сначала взглянем на пример и запустим его.

Начало работы

Сначала создадим приложение React. В оболочке командной строки перейдите в папку, в которой вы хотите создать свой новый проект, и введите следующую команду:

```
create-react-app ipaddress
```

Нажмите клавишу **Enter**/**Return**, чтобы выполнить эту команду. Через несколько секунд будет создан новый проект React. Нам нужно начать с чистого листа, так что придется удалить много файлов. Во-первых, очистите папку *public*. Затем удалите все содержимое папки *src*. Не волнуйтесь: скоро вы заполните их необходимыми файлами, начав с HTML-файла.

В папке *public* создайте файл *index.html*. Добавьте в него следующее содержимое:

```
<! DOCTYPE html>
<html>

<head>
  <title>IP-адрес</title>
</head>

<body>
  <div id="container">

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

В этом коде создается элемент `div` с именем `container`. Затем перейдите в папку *src* и создайте файл *index.js*. В этом файле добавьте следующий код:

```
import React from "react";
import ReactDOM from "react-dom";
import "./index.css";
import IPAddressContainer from "./IPAddressContainer";

var destination = document.querySelector("#container");

ReactDOM.render(
  <div>
    <IPAddressContainer/>
  </div>,
  destination
);
```

Это входная точка для приложения. Код содержит ссылки на шаблоны React, ReactDOM, несуществующий CSS-файл и несуществующий компонент `IPAddressContainer`. Также указан вызов `ReactDOM.render`, который отвечает за запись нашего контента в элемент `div` с именем `container`, который мы определили в HTML-файле пару минут назад.

Перед тем, как перейти к действительно интересному материалу, нужно сделать еще кое-что. В папке *src* создайте файл *index.css* и добавьте в него следующее правило стиля:

```
body {
  background-color: #FFCC00;
}
```

Сохраните все изменения. Мы положили начало приложению. В следующем разделе мы наделим приложение функционалом.

Получение IP-адреса

Далее нам нужно создать компонент, задача которого — получение IP-адреса от веб-сервиса, сохранение его в виде **состояния**, а затем предоставление этого состояния в качестве **свойства** любому компоненту, который его затребует. Давайте создадим необходимый компонент. В папке *src* создайте файл с именем *IPAddressContainer.js*, а затем добавьте в него следующие строки кода:

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

class IPAddressContainer extends Component {
  render() {
    return (
      <p>Вот и все!</p>
    );
  }
}

export default IPAddressContainer;
```

Строки кода, которые вы только что добавили, ничего кардинально не меняют. Они лишь выводят слова «Вот и все!» на экран. На данный момент это неплохо, но давайте продолжим и изменим код, чтобы создать HTTP-запрос. Добавим следующий код:

```
var xhr;

class IPAddressContainer extends Component {
  constructor(props) {
    super(props);

    this.state = {
      ip_address: "…"
    };
```

```
    this.processRequest = this.processRequest.bind(this);
  }

  componentDidMount() {
    xhr = new XMLHttpRequest();
    xhr.open("GET", "https://ipinfo.io/json", true);
    xhr.send();

    xhr.addEventListener("readystatechange",
    this.processRequest, false);
  }

  processRequest() {
    if (xhr.readyState === 4 && xhr.status === 200) {
      var response = JSON.parse(xhr.responseText);

      this.setState({
        ip_address: response.ip
      });
    }
  }

  render() {
    return (
      <div>Вот и все!</div>
    );
  }
};
```

Что-то начинает вырисовываться! Когда компонент становится активным и вызывается метод жизненного цикла `componentDidMount`, мы создаем HTTP-запрос и отправляем его на веб-сервис **ipinfo.io**:

```
.
.
.
componentDidMount() {
  xhr = new XMLHttpRequest();
  xhr.open('GET', "https://ipinfo.io/json", true);
  xhr.send();

  xhr.addEventListener("readystatechange",
  this.processRequest, false);
}
.
.
.
```

Когда мы получаем ответ от службы **ipinfo.io**, вызывается функция `processRequest`, чтобы помочь нам разобраться с результатом:

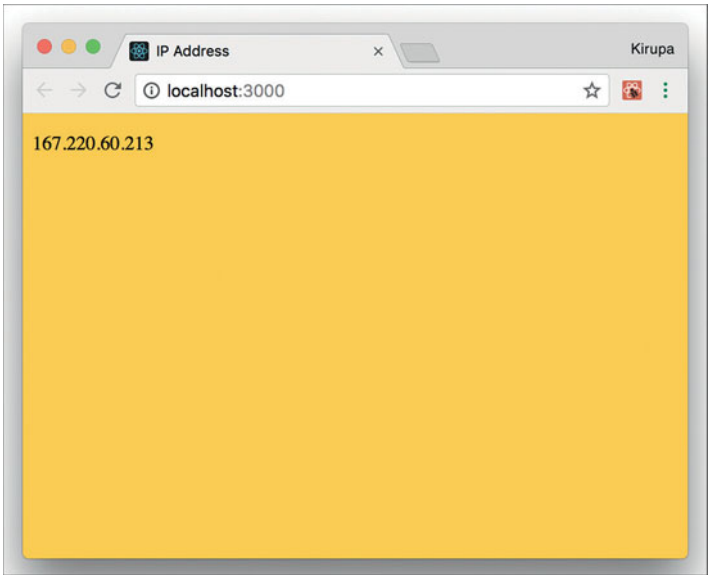
```
.
.
.
processRequest() {
  if (xhr.readyState === 4 && xhr.status === 200) {
    var response = JSON.parse(xhr.responseText);

    this.setState({
      ip_address: response.ip
    });
  }
.
.
.
```

Измените вызов `render` на значение IP-адреса, сохраненное в состоянии:

```
var xhr;

class IPAddressContainer extends Component {
  constructor(props) {
    super(props);

    this.state = {
      ip_address: "…"
    };

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

  componentDidMount() {
    xhr = new XMLHttpRequest();
    xhr.open("GET", "https://ipinfo.io/json", true);
    xhr.send();

    xhr.addEventListener("readystatechange",
    this.processRequest, false);
  }

  processRequest() {
    if (xhr.readyState === 4 && xhr.status === 200) {
      var response = JSON.parse(xhr.responseText);

      this.setState({
        ip_address: response.ip
      });
    }
  }

  render() {
    return (
      <div>{this.state.ip_address}</div>
```

```
        );
      }
    }
```

Если вы запустите приложение в браузере, то увидите IP-адрес. Напоминаю, что вы можете запустить приложение, перейдя в оболочке командной строки в папку *ipaddress* и выполнив команду `npm start`. Когда приложение запустится, оно будет выглядеть примерно так:



Приложение в настоящее время выглядит несколько не так, как хотелось бы, и мы исправим это в следующем разделе.

Визуальные эффекты уровнем выше

Самая трудная часть позади! Мы создали компонент, который обрабатывает все HTTP-запросы, и знаем, что он возвращает IP-адрес при вызове. Теперь мы немного отформатируем вывод, чтобы он выглядел не так аскетично.

Для этого мы не будем добавлять HTML-элементы и прочие детали стилизации в метод `render` компонента `IPAddressContainer`. Вместо этого создадим новый компонент, у которого одна цель — форматирование вывода.

Создайте файл с именем *IPAddress.js* в папке *src*. Затем отредактируйте его, добавив в него следующее содержимое:

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

class IPAddress extends Component {
  render() {
    return (
      <div>
        Привет!
      </div>
    );
  }
}

export default IPAddress;
```

Здесь мы определяем новый компонент с именем `IPAddress`, который будет отвечать за отображение дополнительного текста и обеспечение того, чтобы IP-адрес был отформатирован, как мы хотим. Сейчас это не так, но ситуация очень быстро изменится.

Сначала нужно изменить код метода `render` этого компонента следующим образом:

```
class IPAddress extends Component {
  render() {
    return (
      <div>
        <h1>{this.props.ip}</h1>
        <p>(Это ваш IP-адрес… Наверное: P)</p>
      </div>
```

```
      );
    }
  }

  export default IPAddress;
```

Выделенные изменения должны быть понятны. Мы помещаем результаты значения свойства `ip` в элемент `h1` и выводим некоторый дополнительный текст, используя элемент `p`. Кроме того, добавив в визуализируемый HTML-код немного семантики, мы сможем лучше его стилизовать.

Чтобы отформатировать эти элементы, создайте в папке *src* CSS-файл с именем *IPAddress.css*. В этом файле укажите следующие правила стиля:

```
h1 {
    font-family: sans-serif;
    text-align: center;
    padding-top: 140px;
    font-size: 60px;
    margin: -15px;
}
p {
    font-family: sans-serif;
    color: #907400;
    text-align: center;
}
```

Определив стили, нам нужно сослаться на этот CSS-файл в файле *IPAddress.js*. Для этого добавьте следующую выделенную строку:

```
import React, {Component} from "react";
import "./IPAddress.css";

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

```
      return (
        <div>
          <h1>{this.props.ip}</h1>
          <p>(Это ваш IP-адрес… Наверное: P)</p>
        </div>
      );
    }
  }

  export default IPAddress;
```

Осталось только использовать компонент `IPAddress` и передать IP-адрес. Первый шаг — убедиться, что компонент `IPAddressContainer` знает о компоненте `IPAddress`, ссылаясь на него. В верхней части файла *IPAddressContainer.js* добавьте следующую выделенную строку:

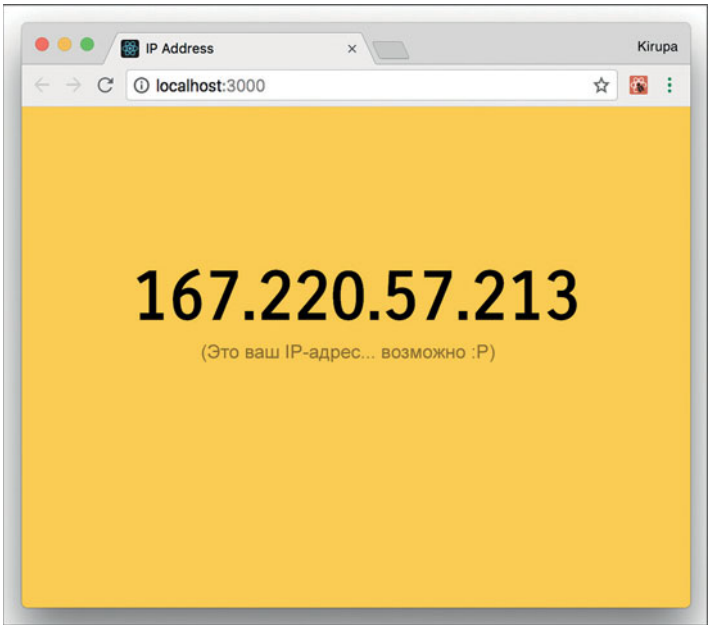
```
  import React, {Component} from "react";
  import IPAddress from "./IPAddress";
      .
      .
      .
```

Второй (и последний!) шаг — изменить код метода `render` следующим образом:

```
      .
      .
      .
    render() {
      return (
        <IPAddress ip={this.state.ip_address}/>
      );
    }
  }
```

С помощью выделенной строки мы вызываем компонент `IPAddress`, определяем свойство `ip` и присваиваем его значение переменной состояния `ip_address`. Это делается для того, чтобы значение IP-адреса все время возвращалось к методу `render` компонента `IPAddress`, где оно форматируется и выводится.

Если вы запустите приложение в своем браузере, вы должны увидеть что-то похожее на пример, показанный в начале главы.



На этом этапе вы закончили с приложением… и почти закончили с этим учебником. Вам нужно знать еще кое-что об этих удивительных компонентах, которые вы добавили.

> Презентационные и контейнерные компоненты
>
> Учитывая все, что мы видели, настало время, чтобы поговорить о выборе дизайна, который мы косвенно затрагивали не только в этом разделе, но и в других. В наших приложениях React мы в основном имеем дело с двумя типами компонентов:

1. **Компоненты, которые отвечают за внешний вид**. Они более известны как презентационные компоненты.

2. **Компоненты, которые выполняют некоторую обработку.** Примеры этой обработки включают в себя маршрутизацию, увеличение счетчика, выборку данных через HTTP-запрос и т. д. Это контейнерные компоненты.

Размышляя о компонентах с точки зрения того, отображают они что-либо (презентационные) или передают данные другим компонентам (контейнерные), вы можете лучше организовать свое приложение React. Чтобы получить полную информацию, ознакомьтесь со статьей Дэна Абрамова по адресу **medium.com/@dan_abramov/smart-and-dumb-components-7ca2f9a7c7d0**.

Заключение

Что же здесь особенного? Все, что мы делали здесь, это использовали скучный старый API JavaScript внутри компонента, подключали события и выполняли те же задачи, что и раньше. И вот что: вы узнали почти все, что нужно, об основах React. Все, о чем мы будем говорить дальше, будет знакомо. Единственные новшества, которые мы рассмотрим, относятся скорее к переопределению и новым формулировкам базовых понятий, которые вы уже знаете, замене их на более новые и профессиональные. В конце концов, разве это не программирование?

Примечание: Если у вас возникнут какие-либо проблемы, задайте вопрос!

Если у вас есть какие-либо вопросы или код не работает ожидаемым образом, не стесняйтесь задать вопрос! Опубликуйте свой вопрос на форуме **forum.kirupa.com** и получите помощь от самых дружелюбных и знающих людей в Интернете!

Глава 15
Создание планировщика в React

Если при создании примера «Привет, мир!» мы радовались тому, что сделали это с помощью React, то в этом случае, создавая приложение-планировщик, будем праздновать получение навыков мастера React. В этой главе мы свяжем воедино много понятий и методов, которые вы изучили, чтобы создать приложение, запущенное на странице **www.kirupa.com/react/examples/todo.htm**.

Все начинается с пустого приложения, которое позднее позволит планировать задачи (см. рис. 15.1).

Рис. 15.1. Пустое приложение с полем ввода

Приложение-планировщик устроено довольно просто. Вы вводите текст задачи или любой другой в поле ввода, а затем нажимаете кнопку **Добавить**. После того, как вы отправите форму, вы увидите новую запись с текстом задачи. Вы можете добавлять новые элементы и отображать их в виде списка (см. рис. 15.2).

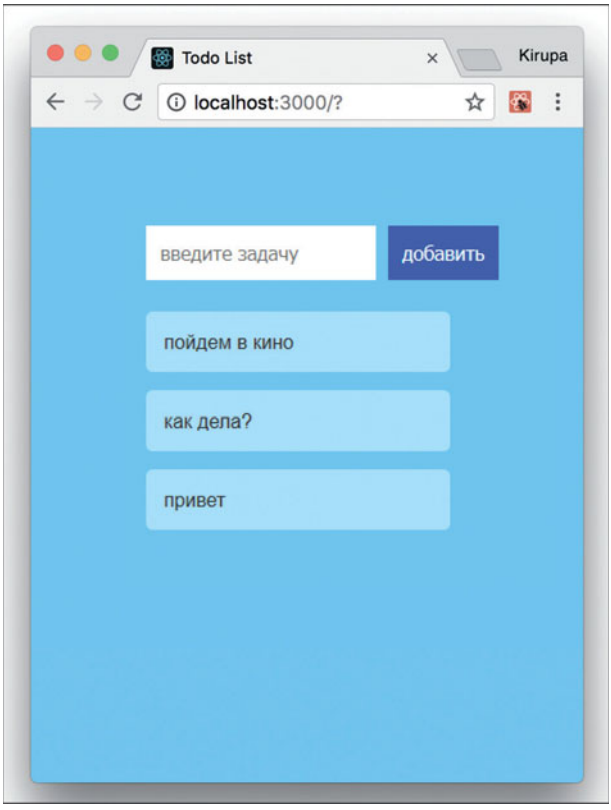
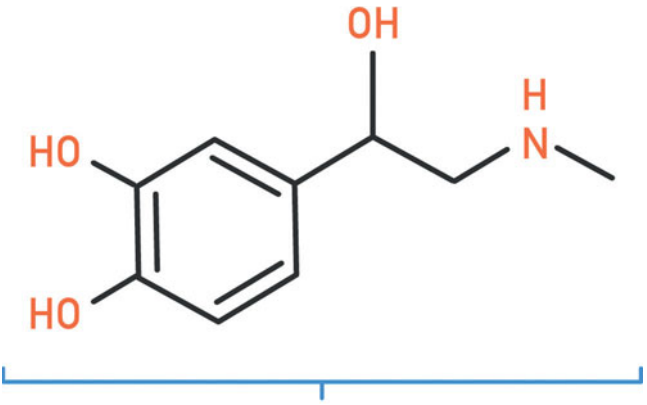


Рис. 15.2. Вы можете добавлять и отображать задачи

Чтобы удалить элемент, щелкните мышью по существующей записи. Довольно просто, не так ли? В следующих разделах, вырабатывая адреналин, мы начнем собирать это приложение с нуля, используя множество восхитительных методов, которые вы изучили ранее.



Химическая структура адреналина (он же эпинефрин)

Это полезное упражнение по разработке каждой части приложения с глубоким вниканием в каждую деталь его работы.

Поехали!

Первый шаг — создать новое приложение React, как это описано в главе 13. В оболочке командной строки перейдите в папку, в которой вы хотите создать свой новый проект, и введите следующую команду:

```
create-react-app todolist
```

Нажмите клавишу **Enter/Return**, чтобы выполнить эту команду. Несколько мгновений спустя новый проект React будет создан. Мы начнем с чистого листа, поэтому удалите все, что содержится в папках *public* и *src*.

К настоящему времени вы все это уже знаете. Нужно с чего-то начать, поэтому в папке *public* создайте HTML-документ *index.html*. В него добавьте следующий контент:

```
<! DOCTYPE html>
<html>
```

```
<head>
  <title>Планировщик</title>
</head>

<body>
  <div id="container">

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

Как вы видите, эта страница довольно проста. Настоящая магия будет происходить в каталоге *src*, где будут находиться JavaScript и CSS-файлы. В каталоге *src* создайте файл *index.css* и добавьте в него следующие правила стиля:

```
body {
  padding: 50px;
  background-color: #66CCFF;
  font-family: sans-serif;
}
#container {
  display: flex;
  justify-content: center;
}
```

Теперь добавьте сценарий JavaScript, который завершит начальную страницу. В том же каталоге *src* создайте файл *index.js*. В этот файл добавьте следующее содержимое:

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

var destination = document.querySelector("#container");
```

```
ReactDOM.render(
  <div>
    <p>Привет!</p>
  </div>,
  destination
);
```

Выделите минутку и посмотрите, что вы добавили. К настоящему времени вы должны полностью понимать, что происходит в коде HTML, CSS и JavaScript. Основа готова. В следующих разделах мы создадим на ней все остальные элементы, составляющие наш планировщик.

Создание интерфейса

На данном этапе функционал нашего приложения невелик. Мы проработаем его чуть позже, а сначала давайте разберем различные элементы пользовательского интерфейса. В нашем случае это несложно. Сначала создадим поле ввода и кнопку. Все это делается с помощью элементов `div`, `form`, `input` и `button`.

Весь код будет находиться внутри компонента `TodoList`. В папке *src* создайте файл *TodoList.js*. Откройте этот файл и добавьте в него следующий код:

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

class TodoList extends Component {
  render() {
    return (
      <div className="todoListMain">
        <div className="header">
          <form>
            <input placeholder="введите задачу">
            </input>
            <button type="submit">ok</button>
```

```
            </form>
          </div>
        </div>
      );
    }
  }
  export default TodoList;
```

Найдите минутку, чтобы взглянуть на добавленный код. Вы увидите много разметки JSX, которая отвечает за заполнение и работу элементов формы. Чтобы использовать недавно созданный компонент `TodoList`, вернемся к файлу *index.js* и сошлемся на него, а затем посмотрим, как выглядит наше приложение. Выполните следующие два изменения:

```
import React from "react";
import ReactDOM from "react-dom";
import "./index.css";
import TodoList from "./TodoList";

var destination = document.querySelector("#container");

ReactDOM.render(
  <div>
    <TodoList/>
  </div>,
  destination
);
```

Сохраните изменения и просмотрите результат в своем браузере. Если все сделано правильно, вы увидите что-то похожее на рис. 15.3.

В приложении есть поле ввода и кнопка отправки формы. Эти два элемента пользовательского интерфейса выглядят не очень привлекательно. Мы исправим это, но сначала поговорим о том, как добавить остальные функции приложения.

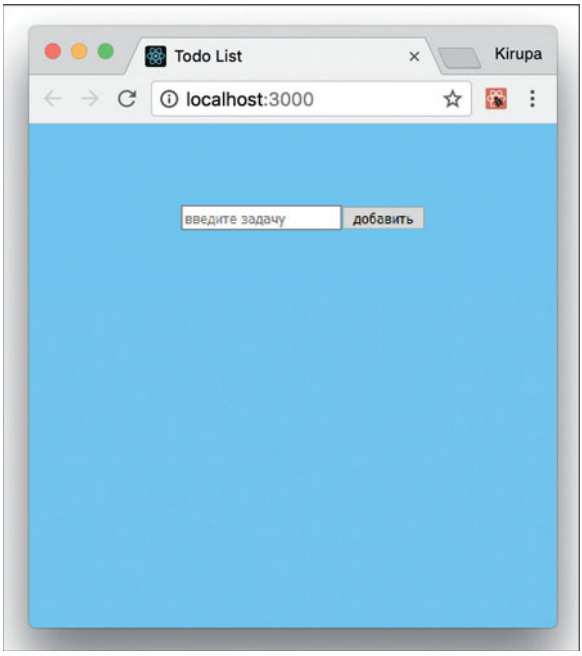


Рис. 15.3. Так выглядит наше приложение прямо сейчас

Построение остальной части приложения

Как вы понимаете, размещение элементов пользовательского интерфейса — это легко. А вот связывание всех визуальных данных с необходимыми данными — настоящий труд. Эту задачу можно условно разделить на пять этапов:

1. Добавление элементов.

2. Отображение элементов.

3. Добавление форматирования.

4. Удаление элементов.

5. Анимация элементов в момент добавления или удаления.

По отдельности все эти этапы реализации просты в написании. Когда вы складываете их вместе, вам нужно следить за несколькими моментами. Мы рассмотрим это и многое другое в следующих разделах.

Добавление элементов

Первая основная задача — настройка обработчиков событий и обработки формы по умолчанию, чтобы мы могли добавить элемент. Вернитесь к элементу `form` и внесите следующие изменения (выделены цветом):

```
class TodoList extends Component {
  render() {
    return (
      <div className="todoListMain">
        <div className="header">
          <form onSubmit={this.addItem}>
            <input placeholder="введите задачу">
            </input>
            <button type="submit">ok</button>
          </form>
        </div>
      </div>
    );
  }
}
```

Мы отслеживаем событие `submit` в самой форме и вызываем метод `addItem`, когда это событие наступило. Обратите внимание, что мы не прослушиваем какое-либо событие на самой кнопке. Это связано с тем, что кнопка имеет атрибут `type` со значением `submit`. Это один из тех трюков HTML, при которых нажатие кнопки с атрибутом `type="submit"` эквивалентно запуску события `submit` в форме.

Теперь пришло время создать обработчик событий `addItem`, который вызывается, когда форма отправлена. Внесите следующие изменения (выделены цветом) чуть выше определения функции `render`:

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

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

  addItem(e) {

  }
    .
    .
    .
}
```

Все, что мы сделали, это определили обработчик события `addItem` и обеспечили правильное определение ключевого слова. Мы до сих пор не сделали ничего, даже отдаленно похожего на добавление задачи, поэтому давайте начнем с первого определения объекта `state` в конструкторе:

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

  this.state = {
    items: []
  };

  this.addItem = this.addItem.bind(this);
}
```

Объект `state` не очень сложный. Мы лишь определяем массив/свойство `items`, который будет отвечать за хранение различных элементов, которые вы можете ввести. Все, что осталось сделать, это считать введенное значение из элемента `input` и сохранить его в массиве `items`, когда пользователь отправит его. Единственная сложность — это фактически чтение значения из элемента DOM. Не очень хорошо, когда мы получаем доступ к элементам DOM и назначаем им свойства, но это дает нам лазейку через ссылки, которые мы можем использовать.

В функции `render` внесите следующие изменения (выделены цветом):

```
render() {
  return (
    <div className="todoListMain">
      <div className="header">
        <form onSubmit={this.addItem}>
          <input ref={(a) => this._inputElement = a}
           placeholder="введите задачу">
          </input>
          <button type="submit">ok</button>
        </form>
      </div>
    </div>
  );
}
```

В листинге показано, что мы сохраняем ссылку на элемент `input` в соответствующим образом названном свойстве `_inputElement`. Теперь мы можем получить доступ к элементу `input`, обратившись к свойству `_inputElement`. Далее настало время заполнить функцию `addItem` следующим содержимым:

```
addItem(e) {
  var itemArray = this.state.items;
```

```
    if (this._inputElement.value!== "") {
      itemArray.unshift({
        text: this._inputElement.value,
        key: Date.now()
      });

      this.setState({
        items: itemArray
      });

      this._inputElement.value = "";
    }

    console.log(itemArray);

    e.preventDefault();
  }
```

Потратьте минутку и поймите что мы делаем. Мы создаем переменную с именем `itemArray` для хранения текущего значения объекта `state` элементов. Затем мы проверяем, содержит ли элемент ввода какое-либо содержимое. Если он пуст, мы ничего не делаем. Если элемент ввода содержит некоторый текст, мы добавляем этот текст к элементу `ItemArray`:

```
itemArray.unshift({
  text: this._inputElement.value,
  key: Date.now()
});
```

Мы не просто добавляем введенный текст. Мы фактически добавляем объект, который содержит как введенный текст, так и уникальное значение ключа, установленное в соответствии с текущим временем (`Date.now()`). Если вы не знаете, почему мы указываем ключ, все в порядке. Я все расскажу сейчас.

Остальная часть кода довольно скучна. Мы присваиваем свойству `items` состояния значение `itemArray`. Мы очищаем значение элемента `input`, чтобы освободить место для следующего ввода. Эта строчка кода менее скучная:

```
e.preventDefault();
```

Мы переопределяем дефолтное поведение этого события. Причина связана с тем, как работают формы. По умолчанию после отправки формы страница перезагружается и очищает все. Нам это не нужно. Вызывая функцию `preventDefault`, мы блокируем поведение по умолчанию. Это хорошая новость!

Пришло время подвести итоги, на каком этапе мы находимся сейчас. Если вы просматриваете приложение и проверяете консоль браузера, то увидите, что объект `state` правильно заполняется с каждым новым добавленным элементом `todo` (см. рис. 15.4).

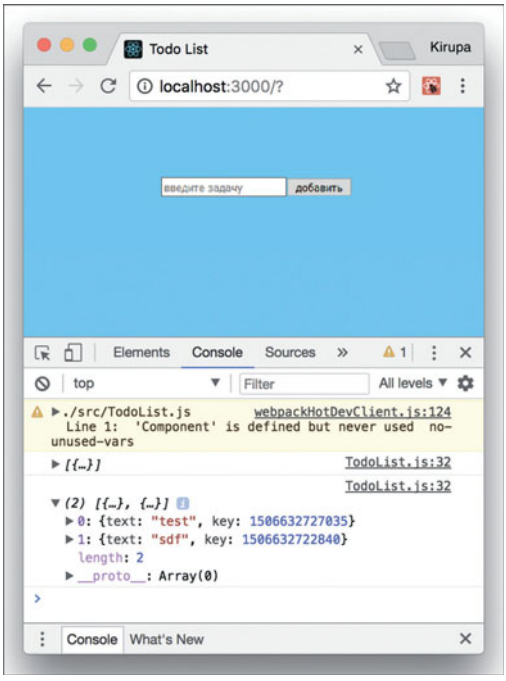


Рис. 15.4. Теперь мы можем видеть сохраненные записи. Это не так уж много, но прогресс очевиден! Без шуток!

> Примечание.
> Подход, альтернативный установке нового состояния внутри функции `addItem`, описан на странице **bit.ly/setStateConcat**.

Отображение элементов

Вывод элементов `todo` исключительно в консоли интересен лишь продвинутым пользователям, поэтому я уверен, что вы захотите отобразить эти элементы непосредственно на странице. Для этого мы возьмем другой компонент. Для начала назовем этот компонент `TodoItems`, определим его в методе `render` компонента `TodoList` и передадим в массив `items` в качестве свойства. Код выглядит следующим образом:

```
render() {
  return (
    <div className="todoListMain">
      <div className="header">
        <form onSubmit={this.addItem}>
          <input ref={(a) => this._inputElement = a}
            placeholder="введите задачу">
          </input>
          <button type="submit">ok</button>
        </form>
      </div>
      <TodoItems entries={this.state.items}/>
    </div>
  );
}
```

После того, как вы это сделали, добавьте в начало документа инструкцию `import`:

```
import React, {Component} from "react";
import TodoItems from "./TodoItems";
```

```
class TodoList extends Component {
  .
  .
  .
```

Это два последних изменения в файле *TodoList.js*. Теперь *создадим* компонент `TodoItems`. В каталоге *src* создайте файл с именем *TodoItems.js* и добавьте в него следующее содержимое:

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

class TodoItems extends Component {
  constructor(props) {
    super(props);

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

  createTasks(item) {
    return <li key={item.key}>{item.text}</li>
  }

  render() {
    var todoEntries = this.props.entries;
    var listItems = todoEntries.map(this.createTasks);

    return (
      <ul className="theList">
        {listItems}
      </ul>
    );
  }
};

export default TodoItems;
```

Выглядит внушительно, потому что мы сразу добавили весь код. Проанализируйте код. В функции `render` мы берем список элементов задач и превращаем их в элементы JSX/HTML. Это производится с помощью вызова функции `map` для элементов и функции `createTasks`:

```
createTasks(item) {
  return <li key={item.key}>{item.text}</li>
}
```

Значение, хранящееся в переменной `listItems`, представляет собой массив элементов `li`, содежащий соответствующий контент для вывода. Обратите внимание, что мы устанавливаем атрибут `key`, значение которому, как вы помните, мы присвоили ранее, применив для каждого элемента функцию `Date.now()`, чтобы упростить для React отслеживание элементов.

Мы превращаем этот список элементов в контент, который можно отобразить на экране:

```
return (
  <ul className="theList">
    {listItems}
  </ul>
);
```

После того, как вы внесете это изменение, сохраните файл и запустите приложение (команда `npm start`, если приложение еще не было запущено). Если все работает правильно, вы сможете не только добавлять элементы, но и просматривать их (см. рис. 15.5).

Если то, что вы видите в браузере, похоже на рисунок ниже, — потрясающе! Теперь сделаем небольшой перерыв и рассмотрим код JS и JSX.

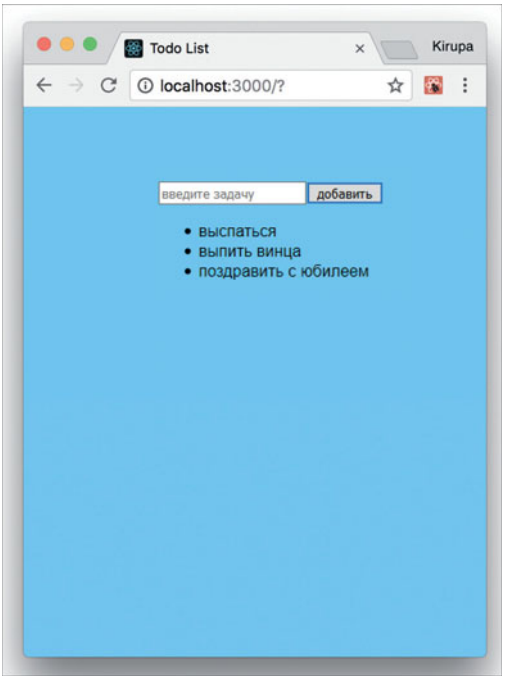


Рис. 15.5. Задачи теперь отображаются на странице!

Форматирование контента приложения

На данном этапе потрясающая функциональность нашего приложения куда лучше, чем его внешний вид. Мы легко исправим эту недоработку, добавив одну таблицу стилей и введя в нее все соответствующие правила стиля. В папке *src* создайте CSS-файл с именем *TodoList.css* и добавьте в него следующие правила стиля:

```
.todoListMain.header input {
  padding: 10px;
  font-size: 16px;
  border: 2px solid #FFF;
  width: 165px;
}
```

```
.todoListMain.header button {
  padding: 10px;
  font-size: 16px;
  margin: 10px;
  margin-right: 0px;
  background-color: #0066FF;
  color: #FFF;
  border: 2px solid #0066FF;
}
.todoListMain.header button: hover {
  background-color: #003399;
  border: 2px solid #003399;
  cursor: pointer;
}
.todoListMain.theList {
  list-style: none;
  padding-left: 0;
  width: 250px;
}
.todoListMain.theList li {
  color: #333;
  background-color: rgba(255,255,255,.5);
  padding: 15px;
  margin-bottom: 15px;
  border-radius: 5px;
}
```

Когда вы создали эту таблицу стилей, вам необходимо сослаться на нее. В верхней части файла *TodoList.js* добавьте ссылку на эту таблицу стилей:

```
import React, {Component} from "react";
import TodoItems from "./TodoItems";
import "./TodoList.css";
```

```
class TodoList extends Component {
  .
  .
  .
```

Если вы просмотрите свое приложение после этого изменения, оно будет выглядеть, как показано на рис. 15.6

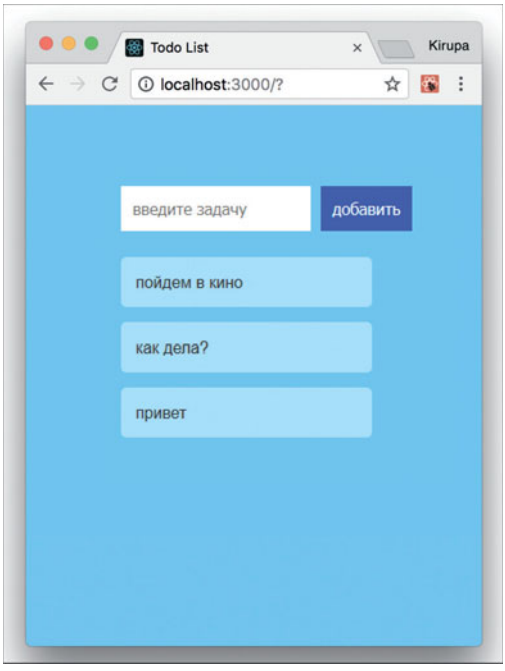


Рис. 15.6. Приложение выглядит намного приятнее

Как вы видите, приложение выглядит намного интереснее. Все, что мы сделали, это добавили CSS-код, поэтому с точки зрения функциональности ничего не изменилось. Дальше мы достигнем большего прогресса в функциональности.

Удаление элементов

На данном этапе мы можем добавлять элементы и отображать их. Но мы не можем удалять элементы после их добавления. Мы разрешим пользователям удалять элементы щелчком мыши по ним. Кажется, что это просто реализовать, не так ли? Единственное, на что нужно обратить внимание, — куда помещать необходимый код. Элементы, по которым планируется щелкать мышью, определяются в файле *TodoItems.js*. Фактическая логика для заполнения элементов находится в объекте `state` в файле *TodoList.js*. Чтобы вы поняли, как нужно поступить, мы воспользуемся некоторыми хитростями, необходимыми, чтобы передать элементы между двумя компонентами.

Сначала нам нужно настроить обработчик события `click`. Измените инструкцию `return` в функции `createTasks` следующим образом:

```
createTasks(item) {
  return <li onClick={() => this.delete(item.key)}
         key={item.key}>{item.text}</li>
}
```

Мы определяем событие `click` и связываем его с обработчиком событий `delete`. Это новый способ передачи аргументов обработчику событий. Из-за того, как аргументы событий и обработчики событий взаимодействуют с областью видимости, мы обходим все эти проблемы с помощью стрелочной функции, которая позволяет использовать значение аргумента по умолчанию и передавать свои собственные аргументы. Если вам кажется это странным, скажу, что это причуда JavaScript, которая не имеет ничего общего с React.

После того, как вы внесете это изменение, вам нужно определить обработчик события `delete`. Внесите следующие изменения (выделены цветом) в код:

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

```
    this.createTasks = this.createTasks.bind(this);
  }

  delete(key) {
    this.props.delete(key);
  }
    .
    .
    .
```

Здесь мы определяем функцию с именем `delete`, которая принимает аргумент для ключа элемента. Чтобы убедиться в корректности кода, мы явно связываем это в конструкторе. Обратите внимание, что функция `delete` фактически ничего не удаляет. Она лишь вызывает *другую* функцию `delete`, переданную в этот компонент через свойства. Здесь мы рассмотрим код более подробно.

В файле *TodoList.js* взгляните на функцию `render`. При вызове компонента `TodoItems` укажем свойство с именем `delete` и установим его в качестве значения функции `deleteItem`:

```
render() {
  return (
    <div className="todoListMain">
      <div className="header">
        <form onSubmit={this.addItem}>
          <input ref={(a) => this._inputElement = a}
             placeholder="введите задачу">
          </input>
          <button type="submit">ok</button>
        </form>
      </div>
      <TodoItems entries={this.state.items}
            delete={this.deleteItem}/>
    </div>
  );
}
```

Это изменение гарантирует, что компонент `TodoItems` теперь оповещен о свойстве `delete`. Это также означает, что функция `delete`, которую мы добавили в компонент `TodoList`, действительно подключена. Все, что остается, определить функцию `deleteItem` так, чтобы она могла удалять элементы.

Сначала добавьте функцию `deleteItem` в компонент `TodoList`:

```
deleteItem(key) {
  var filteredItems = this.state.items.filter(function(item) {
    return (item.key!== key);
  });

  this.setState({
    items: filteredItems
  });
}
```

Вы можете добавить ее в любой позиции, но я рекомендую указать код чуть ниже, где находится функция `addItem`. Проанализируйте, что делает этот код. Мы передаем здесь ключ из элемента, по которому щелкнули мышью, и сверяем этот ключ со всеми элементами, которые были сохранены, с помощью метода `filter`:

```
var filteredItems = this.state.items.filter(function(item) {
  return (item.key!== key);
});
```

Результат работы этого кода прост. Мы создаем новый массив с именем `filtersItems`, который содержит все элементы, кроме удаляемого. Этот отфильтрованный массив затем устанавливается как свойство `items` объекта `state`:

```
this.setState({
  items: filteredItems
});
```

Пользовательский интерфейс обновляется, и элемент удаляется безвозвратно. Последнее, что нам нужно сделать, это разобраться с некоторыми тонкостями. Внесите следующие изменения в код конструктора:

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

  this.state = {
    items: []
  };

  this.addItem = this.addItem.bind(this);
  this.deleteItem = this.deleteItem.bind(this);
}
```

Эта строка гарантирует, что все ссылки в функции `deleteItem` будут корректны. Теперь у нас осталось последнее дело, прежде чем мы сможем объявить победу над удалением предметов. Откройте файл *TodoList.css* и внесите следующие выделенные изменения в код таблицы стилей:

```
.todoListMain.theList li {
  color: #333;
  background-color: rgba(255,255,255,.5);
  padding: 15px;
  margin-bottom: 15px;
  border-radius: 5px;
  transition: background-color .2s ease-out;
}

.todoListMain.theList li: hover {
  background-color: pink;
  cursor: pointer;
}
```

Так вы создадите эффект наведения при помещении указателя мыши поверх элемента, который хотите удалить. После этого изменения работа по удалению элементов завершена. Запустите приложение и попробуйте добавить элементы, а затем удалить их. Все должно работать.

Но есть еще один нюанс …

Анимация! Анимация! Анимация!

Последняя задача — добавить некоторую анимацию, чтобы добавление и удаление элементов происходило естественнее. React предлагает множество способов анимировать интерфейс. Вы можете использовать традиционные подходы, такие как анимации и переходы CSS, `requestAnimationFrame`, Web Animations API или любую другую популярную библиотеку для создания анимаций, которая вам нравится. Все эти подходы позволят вам сделать очень и очень многое.

Но, когда речь заходит об анимировании существующего элемента, традиционные подходы сталкиваются с некоторыми ограничениями. Это связано с тем, что React полностью обрабатывает жизненный цикл элемента, поскольку тот будет удален из DOM. Мы можем переопределить некоторые из методов жизненного цикла, чтобы перехватить удаление элемента и добавить собственную анимационную логику, но это тупиковый путь. Сейчас он нам не подходит.

К счастью, сообществом разработчиков React было создано несколько легких анимационных библиотек, которые упрощают анимирование процессов добавления и удаления элементов. Одна из таких библиотек, **Flip Move**, не только упрощает работу с анимацией, но и умеет многое другое.

Чтобы использовать эту библиотеку, нам нужно сначала добавить ее в наш проект. Откройте оболочку командной строки и убедитесь, что вы все еще находитесь в каталоге проекта `todolist`, а затем выполните следующую команду:

```
npm i -S react-flip-move
```

Нажмите клавишу **Enter**/**Return**, чтобы скопировать необходимый контент в папку *node_modules* проекта. Это обязательно. После того, как вы это сделали, в файл *TodoItems.js* добавьте следующую инструкцию `import`:

```
import FlipMove from 'react-flip-move';
```

Теперь осталось сообщить компоненту `FlipMove`, что нужно анимировать список элементов. В код функции `render` внесите следующие изменения (выделены цветом):

```
render() {
  var todoEntries = this.props.entries;
  var listItems = todoEntries.map(this.createTasks);

  return (
    <ul className="theList">
      <FlipMove duration={250} easing="ease-out">
        {listItems}
      </FlipMove>
    </ul>
  );
}
```

Мы лишь помещаем элементы `listItems` внутрь компонента `FlipMove` и указываем продолжительность анимации и тип функции `easing`. Если вы сейчас запустите приложение, то увидите, что добавление и удаление элементов не происходит внезапно; эти процессы плавно анимируются.

> Неконтролируемые и контролируемые компоненты
>
> Элементы формы по-своему интересны. Они содержат свое определенное состояние. Например, текстовый элемент может содержать в себе некоторый контент, а в раскрывающемся списке могут быть выбраны определенные пункты. React

касается всего, что связано с состояниями, поэтому ему «не по душе», что элементы формы имеют свой собственный внутренний механизм для хранения состояния. Верный путь состоит в том, чтобы синхронизировать все данные формы внутри компонента React, используя такие события, как onChange. Эти компоненты, которые позволяют React управлять элементами формы, известны как контролируемые компоненты.

Тем не менее это проблема, связанная с выполнением каждым элементом формы синхронизации состояния. Разработчики React тоже это понимают. Обходной путь — ничего не делать. Мы позволяем элементам формы работать со своим собственным состоянием и использовать объект `refs` для доступа к значениям, когда это необходимо. Так мы и поступили в этом примере. Компоненты, которые переносят все управление состоянием на элемент DOM формы, называются **неконтролируемыми компонентами**.

Заключение

Приложение-планировщик функционально довольно простое, но, собрав его с нуля, мы рассмотрели почти каждую деталь, касающуюся React. Самое важное: этот пример демонстрирует, как различные подходы, изученные вами по-отдельности, работают вместе. И это на самом деле важно.

Теперь вот какой вопрос: имеет ли смысл все, что мы сделали в этой главе? Если это так, вы можете рассказать своим друзьям и родным, что близки к освоению React. Если же некоторые моменты непонятны, я рекомендую вернуться и перечитать главы, в которых они рассматриваются.

Примечание: Если у вас возникнут какие-либо проблемы, задайте вопрос!

Если у вас есть какие-либо вопросы или код не работает ожидаемым образом, не стесняйтесь задать вопрос! Опубликуйте свой вопрос на форуме **forum.kirupa.com** и получите помощь от самых дружелюбных и знающих людей в Интернете!

Глава 16
Создание плавающего меню в React

Плавающее меню сегодня — весьма распространенный элемент в пользовательских интерфейсах. Эти меню, по сути, — внеэкранные элементы, которые выплывают после щелчка мыши или касания по некоему элементу на странице. Это может быть стрелка, значок из трех линий или что-то еще, указывающее на меню.

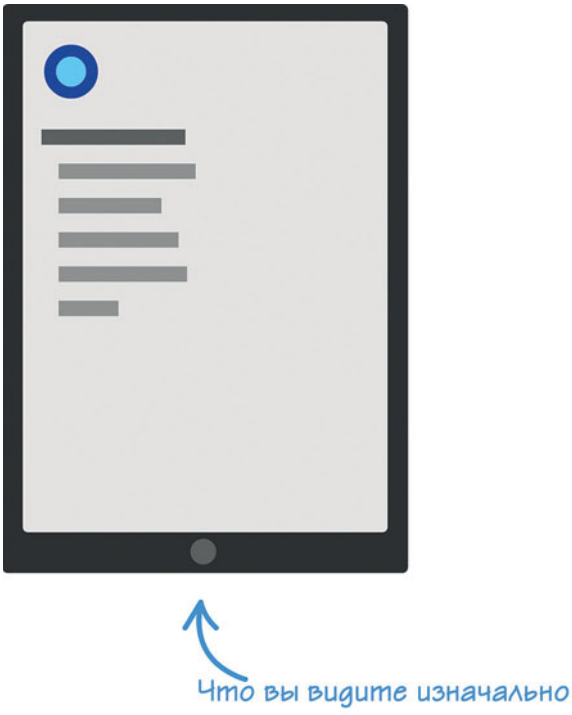
Чтобы увидеть плавающее меню в действии, посетите страницу **www.kirupa.com/react/examples/slidingmenu_css/index.html**.

Если вы щелкнете мышью по кружочку, то увидите выплывающее желтое меню со ссылками. Если вы щелкнете мышью по любой ссылке или даже вне их, меню уплывет назад, и вы вновь увидите контент страницы. Давайте посмотрим, как создать такое меню с помощью React.

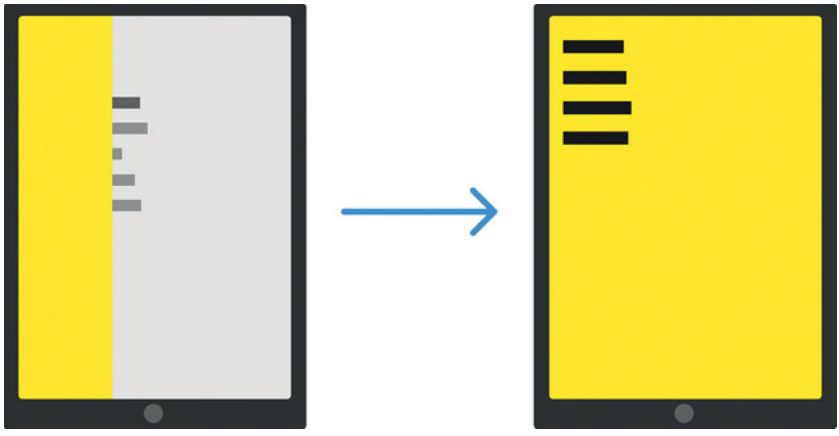
> Примечание. Решение без React.
> Если вы хотите создать это меню с помощью обычного JavaScript-кода без трюков React, прогчитайте мою статью на странице **bit.ly/plainSidingMenu**.

Как работает плавающее меню

Прежде чем разбираться в коде, давайте рассмотрим, как работает плавающее меню. Если с самого начала, то у нас есть страница, на которой отображается некоторый контент:



Если вы решите открыть меню (щелкнув мышью по синему кружку), меню волшебным образом появится на экране:



Как работает это плавающее меню, не так сложно понять. Меню есть всегда; оно просто скрыто за пределами экрана. Чтобы понять, как это работает, взгляните на следующую диаграмму:

Слева от контента мы видим меню, терпеливо скрывающееся до тех пор, пока оно не будет вызвано. Мы скрываем его, перемещая так далеко, насколько можем, до тех пор, пока оно не будет полностью скрыто. Выяснить, насколько нужно его переместить — легко. Размер меню совпадает с размером окна браузера, потому что нам нужно, чтобы меню полностью закрывало страницу. Учитывая эту деталь, мы смещаем меню влево на ширину браузера. Один из способов сделать это — с помощью таблицы стилей, код которой выглядит следующим образом:

```
#theMenu {
  position: fixed;
  left: 0;
```

```
    top: 0;
    transform: translate3d(-100vw, 0, 0);
    width: 100vw;
    height: 100vh;
  }
```

Положение меню зафксировано (`fixed`). Это единственное изменение предоставляет меню множество магических возможностей. Во-первых, гарантирует, что правила обычной компоновки больше не применяются к нему. Мы можем разместить меню в любом месте, указав значения `x` и `y`, и меню не будет смещаться в сторону от указанной позиции. Меню даже не отобразит полосу прокрутки, если мы скроем ее за пределами экрана.

Все это хорошо, потому что мы скрываем меню, присваивая свойствам `left` и `top` меню значение `0`, а свойству `transform` — метод `translate3d` с горизонтальным значением `-100vw`. Отрицательное значение гарантирует, что мы перемещаем меню влево на значение, эквивалентное ширине окна браузера. Важную роль также играет размер меню, не связанный непосредственно с позицией. Вот почему в этом CSS-коде мы используем свойства `width` и `height` со значениями `100vw` и `100vh`, соответственно, чтобы размер меню был таким же, как размер окна браузера.

> Что такое `vw` и `vh`?
> Если вы никогда не видели единицы измерения `vw` и `vh`, то знайте, что они обозначают **viewport width** (ширину экрана) и **viewport height** (высоту экрана). Есть что-то схожее с процентными значениями. Каждый блок имеет ширину или высоту в 1/100 ширины окна просмотра (что мы вызываем в окне браузера). Например, значение `100vw` означает всю ширину окна браузера. Аналогично `100vh` относится к значению всей высоты окна браузера.

Когда меню вызывается для просмотра, мы перемещаем меню до тех пор, пока его горизонтальное положение не будет таким же,

что и начальная позиция окна браузера. Код таблицы стилей для этого действия очень прост. Мы лишь присваиваем метод `translate3d` свойству `transform` и настраиваем горизонтальное положение, указав значение `0vw`.

Код выглядит так:

```
transform: translate3d(0vw, 0, 0);
```

Это изменение гарантирует, что меню теперь видно. Когда меню должно исчезнуть, мы можем вернуть все обратно:

```
transform: translate3d(-100vw, 0, 0);
```

Самое интересное, о чем мы еще не говорили, это анимация, которая делает плавающее меню эффектным. Эффект достигается с помощью простого перехода CSS, анимирующего свойство `transform`:

```
transition: transform .3s cubic-bezier(0, .52, 0, 1);
```

Если вы не знакомы с переходами CSS, с ними нужно держать ухо востро. Я не буду объяснять работу с ними здесь, поэтому выделите пару минут и прочитайте мою краткую статью о переходах на сайте **www.kirupa.com/html5/introduction_css_transitions.htm**.

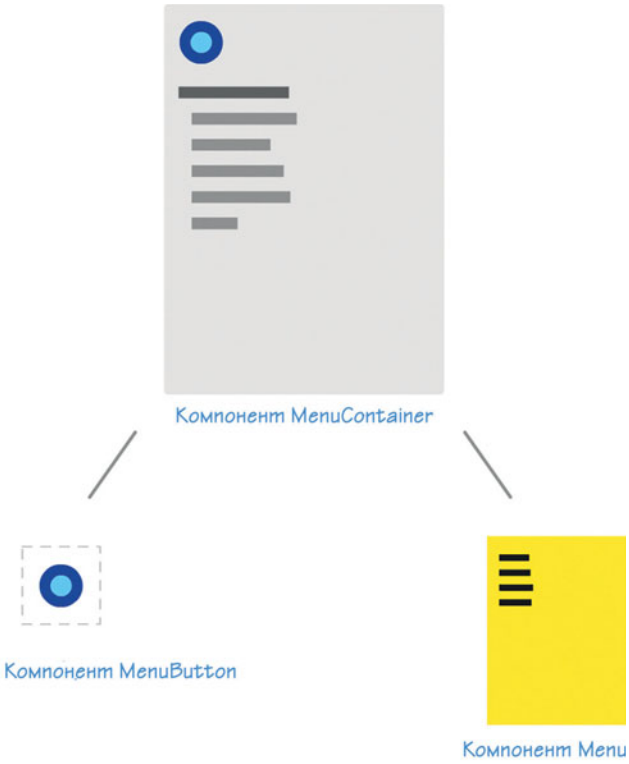
До сих пор мы с высоты смотрели, как работает плавающее меню. Нужно рассмотреть еще несколько нюансов, чем мы займемся в следующих разделах, а пока создадим это меню.

Настройка плавающего меню

Теперь, когда у вас есть общее представление о том, как работает плавающее меню, давайте превратим все эти теоретические знания в необходимый JSX- и JS-код. Первое, что мы сделаем, это взглянем на этот пример с точки зрения отдельных компонентов, которые и будут составлять меню.

На самом верху используется компонент `MenuContainer`:

Этот компонент отвечает за выполнение скрытых операций, таких как управление состоянием, размещение компонентов `Menu` и `MenuButton` и отображение определенного текста. Выглядит все это примерно так:



В следующих разделах мы начнем создавать эти компоненты и запустим пример.

Начало работы

Выполните команду `create-react-app` для создания нового проекта под названием `slidemenu`. Если вы не знаете, как это сделать, ознакомьтесь с главой 13, чтобы просмотреть информацию о создании и работе с проектами React. После того как вы создали проект, вам нужно начать все с чистого листа. Удалите все содержимое папок *public* и *src*. Далее вы заново создадите необходимые файлы приложения.

Начнем с создания HTML-документа. В папке *public* создайте файл с именем *index.html*. В него добавьте следующее содержимое:

```
<! DOCTYPE html>
<html>

<head>
  <title>Плавающее меню в React</title>
</head>

<body>
  <div id="container"></div>
</body>

</html>
```

Эта страница HTML — лишь расположение, в котором все наши компоненты React в конечном итоге будут выводить результат.

Затем в папке *src* создайте файл *index.js*, который будет отвечать за все, что связано с кодом. Добавьте следующий код в этот файл:

```
import React from "react";
import ReactDOM from "react-dom";
import "./index.css";
import MenuContainer from "./MenuContainer";

ReactDOM.render(
  <MenuContainer/>,
  document.querySelector("#container")
);
```

Вызов функции `render` отвечает за отображение вывода компонента `MenuContainer` в элементе `container div`, который мы ранее указали в HTML-файле. В инструкциях `import`, помимо импорта библиотек `react` и `react-dom`, мы ссылаемся на файл *index.css* и компонент `MenuContainer`. Это все, что есть в файле *index.js*.

Затем создайте файл *index.css* в папке *src* и определите базовый стиль страницы. В этом файле добавьте следующие два правила стиля:

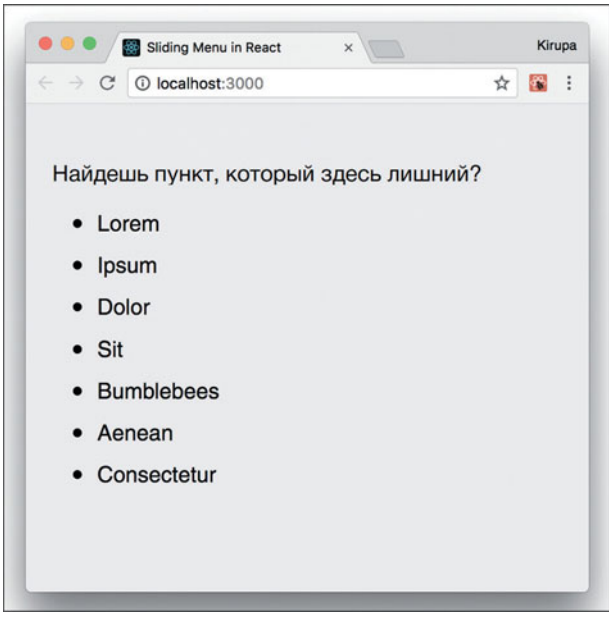
```
body {
  background-color: #EEE;
  font-family: sans-serif;
  font-size: 20px;
  padding: 25px;
  margin: 0;
  overflow: auto;
}

#container li {
  margin-bottom: 10px;
}
```

Здесь особо нечего сказать об этих правилах стиля, поэтому поехали дальше. Для начальной настройки приложения создайте компонент `MenuContainer`. Создайте файл с именем MenuContainer.js в папке *src* и добавьте в него следующий JS- и JSX-код:

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

class MenuContainer extends Component {
  render() {
    return (
      <div>
        <div>
          <p>Найдешь пункт, который здесь лишний?</p>
          <ul>
            <li>Lorem</li>
            <li>Ipsum</li>
            <li>Dolor</li>
            <li>Sit</li>
            <li>Bumblebees</li>
            <li>Aenean</li>
            <li>Consectetur</li>
          </ul>
```

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

export default MenuContainer;
```

Обязательно сохраните изменения, внесенные вами во все файлы, и протестируйте приложение (выполнив команду `npm start`), чтобы убедиться, что первоначальная версия приложения работает нормально. Если все настроено верно, запустится дефолтный браузер, и вы увидите следующую страницу:



Ни меню, на кнопки для его вызова еще нет; мы добавим их в следующих разделах.

Отображение и сокрытие меню

С первоначальной страницей разобрались, пришло время создать меню. Меню отображается/скрывается следующим образом:

1. Когда вы щелкаете мышью по кнопке, меню выплывает для просмотра;

2. Когда вы щелкаете мышью в любой позиции меню, меню уплывает.

Это означает, что нам нужно учитывать некоторые моменты. Нам нужно поддерживать определенное состояние, чтобы отслеживать, скрыто ли меню, либо отображено. Это состояние должно обновляться как с помощью кнопки, так и с помощью меню, потому что щелчок мышью по кнопке будет переключать видимость меню (отображается или скрыто меню). Нам нужно, чтобы состояние располагалось в одной локации, доступной как меню, так и кнопке. Это единое расположение будет находиться внутри компонента `MenuContainer`, поэтому давайте добавим код, относящийся к логике состояния.

В файле *MenuContainer.js* добавьте методы `constructor` и `toggleMenu` чуть выше метода `render`:

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

  this.state = {
    visible: false
  };

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

toggleMenu() {
  this.setState({
    visible:!this.state.visible
  });
}
```

Код, который вы только что добавили, совсем прост. Вы сохраняете переменную с именем `visible` в объекте `state`, и создаете метод `toggleMenu`, который будет нести ответственность за то, какое значение присвоено переменной `visible`: `true` или `false`.

Далее мы рассмотрим события `click` на кнопке и меню. Если целью является обновление состояния компонента `MenuContainer`, то нам нужно также разместить в `MenuContainer` обработчик событий. Внесите следующие изменения в код (выделены цветом):

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

class MenuContainer extends Component {
  constructor(props) {
    super(props);

    this.state = {
      visible: false
    };

    this.handleMouseDown = this.handleMouseDown.bind(this);
    this.toggleMenu = this.toggleMenu.bind(this);
  }

  handleMouseDown(e) {
    this.toggleMenu();

    console.log("clicked");
    e.stopPropagation();
  }

  toggleMenu() {
    this.setState({
      visible:!this.state.visible
    });
```

```
    }
    .
    .
    .
}
```

Когда вызывается метод `handleMouseDown`, мы вызываем `toggleMenu`, который переключает отображение меню. На данный момент вы, вероятно, задаетесь вопросом, где фактический код для работы с событием `click`? Что именно повлечет вызов `handleMouseDown`? Ответ таков: ничего! Мы работали в обратном порядке, и сначала определили обработчик событий. Мы настроим связи между обработчиком событий и событием `click` чуть позже, а пока займемся кнопкой и компонентами меню.

Создание кнопки

В папке *src* создайте два файла с именем *MenuButton.js* и *MenuButton.css*. Затем откройте файл *MenuButton.js* в текстовом редакторе. Добавьте в него следующие строки кода:

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

class MenuButton extends Component {
  render() {
    return (
      <button id="roundButton"
          onMouseDown={this.props.handleMouseDown}></button>
    );
  }
}

export default MenuButton;
```

Разберитесь, что делает этот код. Происходит немногое. Мы определяем элемент кнопки с именем `roundButton` и связываем событие `onMouseDown` с именем свойства, на которое мы ссылаемся как `handleMouseDown`. Прежде чем читать дальше, откройте файл *MenuButton.css* и добавьте следующие правила стиля:

```
#roundButton {
  background-color: #96D9FF;
  margin-bottom: 20px;
  width: 50px;
  height: 50px;
  border-radius: 50%;
  border: 10px solid #0065A6;
  outline: none;
  transition: all .2s cubic-bezier(0, 1.26, .8, 1.28);
}

#roundButton: hover {
  background-color: #96D9FF;
  cursor: pointer;
  border-color: #003557;
  transform: scale(1.2, 1.2);
}

#roundButton: active {
  border-color: #003557;
  background-color: #FFF;
}
```

Теперь настало время сделать экземпляр только что созданного компонента `MenuButton`. Вернитесь к компоненту `MenuContainer` и добавьте следующую выделенную строку кода внутри метода `render`:

```
render() {
  return (
    <MenuButton handleMouseDown={this.handleMouseDown}/>
```

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

Чтобы эта строка действительно что-то делала, обязательно добавьте соответствующую инструкцию `import` в верхней части файла *MenuButton.js*! Это легко пропустить!

Обратите внимание, что мы передаем в свойство `handleMouseDown`, и его значение — обработчик событий `handleMouseDown`, который мы определили ранее. Так гарантируется, что при щелчке мышью по кнопке внутри компонента `MenuButton`, вызывается метод `handleMouseDown`, который находится в компоненте `MenuContainer`. Все это замечательно, но кнопка бесполезна без меню. Давайте решим эту задачу.

Создание меню

Пришло время создать компонент `Menu`, который будет отвечать за все, что касается меню. Прежде чем мы создадим этот компонент, притворимся, что он уже существует и вызывается из метода `render` компонента `MenuContainer`. Добавьте следующий выделенный вызов в код компонента `Menu` (в настоящее время воображаемый) чуть ниже, где вы добавили вызов `MenuButton`:

```
  render() {
    return (
      <MenuButton handleMouseDown={this.handleMouseDown} />
      <Menu handleMouseDown={this.handleMouseDown}
            menuVisibility={this.state.visible} />
      .
      .
      .
    );
  }
```

Добавьте инструкцию `import` файла *Menu.js*. Возвращаясь к компоненту меню, взгляните на свойства, которые вы передаете. Первое свойство должно быть знакомо. Это `handleMouseDown` и его значение — метод обработки событий `handleMouseDown`. Второе свойство носит имя `menuVisibility`. Его значение — текущее значение свойства состояния `visible`. Теперь создадим компонент `Menu` и среди прочего посмотрим, как эти свойства будут использоваться.

В папке *src* создайте файлы *Menu.js* и *Menu.css*. В файл *Menu.js* добавьте следующий код:

```
import React, {Component} from "react";
import "./Menu.css";

class Menu extends Component {
  render() {
    var visibility = "hide";

    if (this.props.menuVisibility) {
      visibility = "show";
    }

    return (
      <div id="flyoutMenu"
            onMouseDown={this.props.handleMouseDown}
            className={visibility}>
        <h2><a href="#">Главная</a></h2>
        <h2><a href="#">About</a></h2>
        <h2><a href="#">Contact</a></h2>
        <h2><a href="#">Search</a></h2>
      </div>
    );
  }
}

export default Menu;
```

Обратите внимание на JSX-код в инструкции `return`. У нас есть элемент `div` с именем `flyoutMenu` и некоторым содержимым. В элементе `div` мы вызываем метод обработки событий `handleMouseDown` (передаваемый как свойство), когда происходит событие `onMouseDown`. Затем мы присваиваем этому элементу значение `class`; значение является результатом оценки переменной `visible`. Как вы знаете, `class` — зарезервированное ключевое слово в языке JavaScript, и вы не можете использовать его в JSX-коде; поэтому я использовал имя `className`.

Возвращаясь к коду, значение свойства `visibility` устанавливается несколькими строками:

```
var visibility = "hide";

if (this.props.menuVisibility) {
  visibility = "show";
}
```

Значение может быть либо `hide`, либо `show`, в зависимости от значения свойства `menuVisibility`, которое может быть (значение задается свойством состояния `visible`) `true` или `false`. На первый взгляд кажется, что это не так, но код, окружающий `className`, играет очень важную роль в определении, действительно ли меню отображается на экране. Если взглянуть на CSS-код, вы поймете почему. Теперь откройте файл *Menu.css* и добавьте в него следующие правила стиля:

```
#flyoutMenu {
  width: 100vw;
  height: 100vh;
  background-color: #FFE600;
  position: fixed;
  top: 0;
  left: 0;
  transition: transform .3s
        cubic-bezier(0, .52, 0, 1);
```

```
  overflow: scroll;
  z-index: 1000;
}

#flyoutMenu.hide {
  transform: translate3d(-100vw, 0, 0);
}

#flyoutMenu.show {
  transform: translate3d(0vw, 0, 0);
  overflow: hidden;
}

#flyoutMenu h2 a {
  color: #333;
  margin-left: 15px;
  text-decoration: none;
}

#flyoutMenu h2 a: hover {
  text-decoration: underline;
}
```

CSS-код, который вы видите здесь, в основном касается внешнего вида меню, а отображение и сокрытие меню обрабатываются правилами `#flyoutMenu.hide` и `#flyoutMenu.show`. Какие из этих правил стиля активны, полностью зависит от кода, который мы рассмотрели ранее. Как вы помните, в элементе `flyoutMenu div` значение свойства `class` в сгенерированном HTML-коде будет либо `hide`, либо `show`, в зависимости от того, какое значение присвоено свойству `className`. Довольно круто, не так ли?

На этом этапе мы закончили с кодом. Обязательно сохраните все изменения и убедитесь, что приложение работает так же, как и мой пример, с которого мы начали. Однако не зацикливайтесь на этом проекте. Дальше мы рассмотрим основные его недостатки.

Заключение

В этой главе мы использовали React для создания популярного элемента пользовательского интерфейса — плавающего меню. В главе вы узнали о взаимодействии между компонентами, такими как работа с событиями/обработчиками событий и т. д. Если рассмотреть другие примеры, вы увидите, что функции React на этом исчерпаны. Все, что остается, — это правильно использовать и менять местами одни и те же концепции, выстраивая более сложные сценарии. Это не значит, что пора отложить книгу. Есть еще несколько проектов React, которые нужно создать и полностью разобрать!

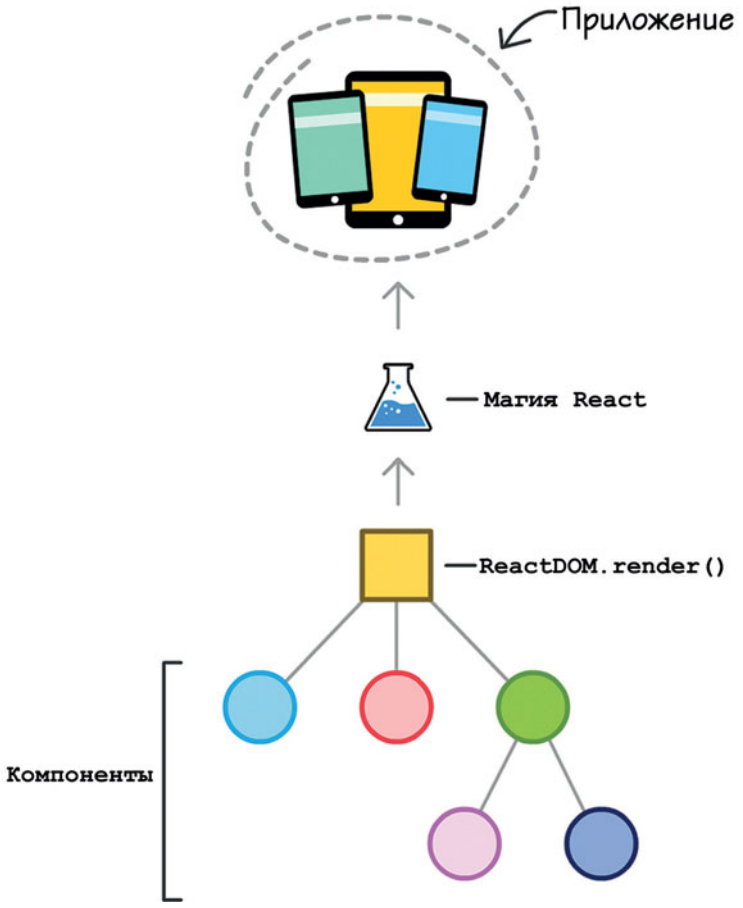
> Примечание: Если у вас возникнут какие-либо проблемы, задайте вопрос!
>
> Если у вас есть какие-либо вопросы или код не работает ожидаемым образом, не стесняйтесь задать вопрос! Опубликуйте свой вопрос на форуме **forum.kirupa.com** и получите помощь от самых дружелюбных и знающих людей в Интернете!

Глава 17
Избегайте ненужных операций рендеринга

Вы, наверное, очень устали от того, что я повторяю это, но быстрая работа с DOM — одно из самых больших преимуществ React. Это не значит, что вы даром получаете отличную производительность. Пока React выполняет много сложных операций, вам нужно предпринять некоторые шаги, чтобы убедиться в оптимизации работы вашего приложения. Один из самых важных шагов на этом пути заключается в проверке, что метод `render` каждого компонента вызывается только тогда, когда это действительно необходимо. В следующих нескольких разделах мы рассмотрим, почему может возникнуть такая проблема и как с ней справиться.

Вкратце о методе `render`

Официальное описание метода `render` довольно простое. Он лишь должен отображаться на каждом компоненте и помогать генерировать JSX-код для возврата к любому родительскому компоненту, вызвавшему его. Если описывать полный рабочий цикл с компонентами на одном конце и полностью завершенным приложением на другом, схема выглядела бы так:



У вас есть готовое приложение с одной стороны, и есть компоненты, составляющие приложение, с другой. Внутри всех этих компонентов находятся методы `render`, возвращающие фрагменты JSX-кода, которые объединяются с фрагментами JSX-кода от других компонентов. Этот процесс повторяется, пока вы не получите финальный JSX-код в корне иерархии компонентов, где есть вызов `ReactDOM.render`. Здесь происходит **магия React**, превращающая весь JSX-код в соответствующий HTML/CSS/JS-код для отображения в браузере.

Теперь, когда у вас есть весьма поверхностное представление о том, как работает React, давайте вернемся к позициям, где находятся компоненты и их методы `render`. Во всех примерах приложений

React, с которыми вы работали до этого момента, вам никогда не приходилось явно вызывать метод `render` на каком-либо компоненте. Это происходило автоматически. Давайте уточним. Три ситуации заставляют автоматически вызывать метод `render`:

1. Свойство компонента обновляется.

2. Свойство состояния компонента обновляется.

Вызывается метод `render` родительского компонента.

Все три ситуации кажутся идеальными для того, чтобы метод `render` компонента автоматически вызывался. В конце концов, все три ситуации *могут* привести к изменению визуального контента, не так ли?

Ответ: Да! Очень часто компоненты вынуждены вызывать метод `render` повторно, хотя переменная или состояние, которое меняется, не имеет к ним никакого отношения. В некоторых ситуациях родительский компонент правильно выполняет метод `render`, но он локализуется только для этого компонента. Нет необходимости запрашивать дочерние компоненты для повторной обработки чего-то, что не влияет на них.

Теперь я мог бы нарисовать тревожную картину ненужной работы, которая шла у нас прямо под носом. Следует иметь в виду, что вызываемый метод `render` — это не то же самое, что DOM, который в конечном счете, обновляется. React выполняет несколько дополнительных действий, в которых DOM различается (то есть предыдущая версия сравнивается с новой/текущей версией), чтобы проверить, нужно ли представлять какие-либо изменения. Все эти «несколько дополнительных действий» означают некую работу, и более сложные приложения с большим количеством компонентов будут сталкиваться с многочисленными проблемами, которые начнут возникать. Некоторые из них — дополнительная работа, выполняемая встроенными средствами React. Некоторые из них — это важные операции, которые мы делаем в методах `render`; у нас часто есть много кода для генерации соответствующего JSX-контента. Редко метод `render` возвращает статическую часть JSX-кода без каких-либо изменений или вычислений, поэтому избавление от ненужных вызовов `render` — это хорошо.

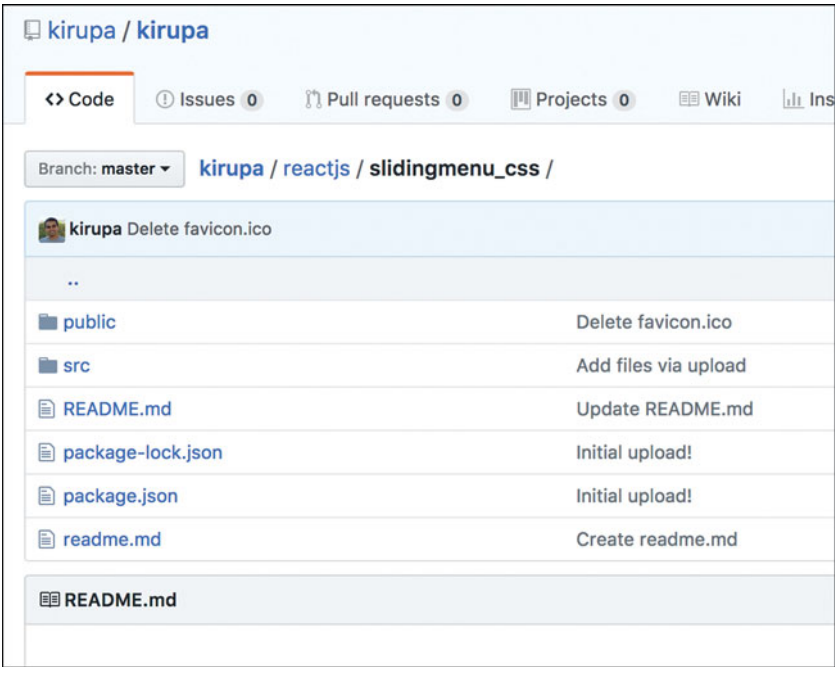
Оптимизация количества вызовов метода `render`

Теперь, когда вы осознали проблему, давайте рассмотрим некоторые подходы, которые можно использовать, чтобы вызывать метод `render` компонента только тогда, когда это действительно необходимо. В следующих разделах я расскажу об этом.

Разбираем пример

Чтобы разобраться в деталях, рассмотрим пример, причем не новый, а наше плавающее меню, которое мы создали в прошлой главе. Для удобства откройте его в редакторе кода.

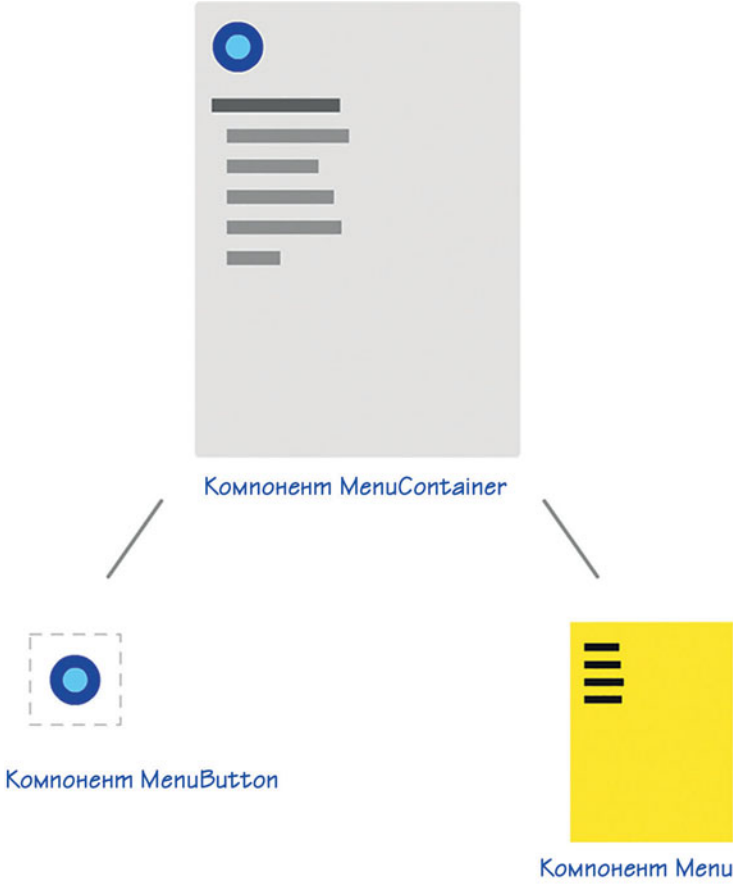
Если у вас нет подходящего проекта, не волнуйтесь. Выполните команду `create-response-app` для создания нового проекта React, удалите все содержимое из папок *src* и *public* и поместите туда соответствующие файлы и папки из моего репозитория, показанного на рисунке ниже:



Когда проект меню будет готов, запустите его в браузере, чтобы проверить, что меню работает.

Если вы не делали плавающее меню при чтении предыдущей главы, я настоятельно рекомендую вам это сделать. Наличие рабочего проекта под рукой — это важно, но еще важнее знать, как работает код, и понимать некоторые из решений, которые мы использовали в процессе реализации. Вы можете читать главу и без примера, но потом не говорите, что тот или иной код покажется непонятным.

Если взглянуть на пример, показанный на рисунке ниже, иерархия компонентов плавающего меню выглядит следующим образом:



В корне находится `MenuContainer`, и у него есть два потомка: `MenuButton` и `Menu`. Хотя это не показано на диаграмме, но существует вызов `ReactDOM.render` в файле *index.js*, который компонент `MenuContainer` предоставляет в DOM:

```
ReactDOM.render(
  <MenuContainer/>,
  document.getElementById("container")
);
```

Когда нажата кнопка, отображаемая с помощью компонента `MenuButton`, мы присваиваем свойству состояния (с именем `visible`) компонента `MenuContainer` логическое значение `true`. Это изменение свойства состояния приводит к тому, что компонент `Menu` обновляет значение класса, которое активирует соответствующие правила CSS, перемещая меню. Щелчок мышью в любой позиции меню скрывает его, отменяя изменения, внесенные ранее, путем присвоения свойству состояния компонента `MenuContainer` значения `false`.

Анализ вызовов метода `render`

Первое, что следует сделать, это просмотреть вызовы метода `render`. Вы можете сделать это разными способами. Например, установить точку останова в своем коде и проверить результаты с помощью инструментов разработчика браузера. Или установить расширение React Developer Tools (для Chrome или Firefox) со страницы **github.com/facebook/react-devtools** и проверить каждый компонент. Вы также можете пойти самым простым путем и вставлять инструкции `console.log` в интересующие вас методы `render`.

Поскольку в плавающемся меню есть только три компонента, подход `console.log` наиболее удобен, и сейчас мы будем его использовать. В редакторе откройте файлы *MenuContainer.js*, *MenuButton.js* и *Menu.js* и прокрутите код до метода `render` каждого компонента.

В верхней части этого метода мы добавим вызов инструкции `console.log`.

В файле *MenuContainer.js* добавьте следующую выделенную строку:

```
render() {
  console.log("Rendering: MenuContainer");
  return (
    <div>
      <MenuButton handleMouseDown={this.handleMouseDown}/>
      <Menu handleMouseDown={this.handleMouseDown}
            menuVisibility={this.state.visible}/>
      <div>
        <p>Can you spot the item that doesn&apos; t belong?</p>
        <ul>
          <li>Lorem</li>
          <li>Ipsum</li>
          <li>Dolor</li>
          <li>Sit</li>
          <li>Bumblebees</li>
          <li>Aenean</li>
          <li>Consectetur</li>
        </ul>
      </div>
    </div>
  );
}
```

Аналогичным образом поступим и с файлом *MenuButton.js*:

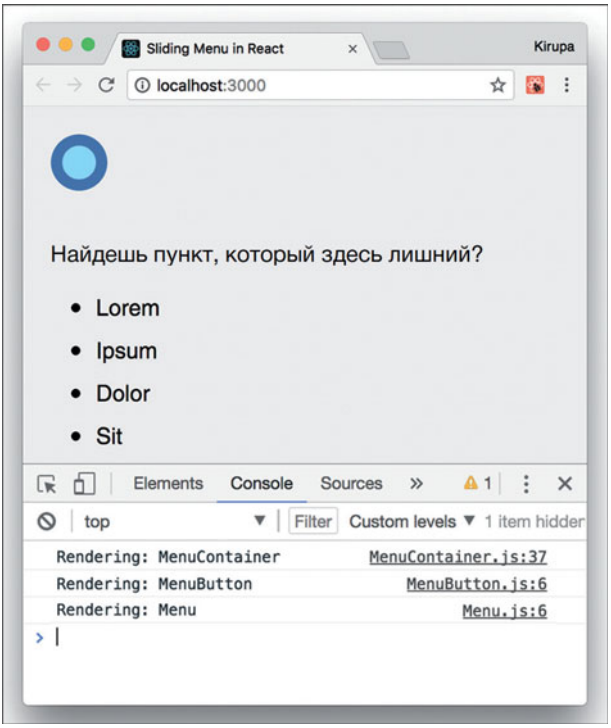
```
render() {
  console.log("Rendering: MenuButton");

  return (
    <button id="roundButton"
        onMouseDown={this.props.handleMouseDown}></button>
```

```
    );
  }
```

И наконец, внесем изменения в файл *Menu.js*:

```
  render() {
    console.log("Rendering: Menu");

    var visibility = "hide";

    if (this.props.menuVisibility) {
      visibility = "show";
    }

    return (
      <div id="flyoutMenu"
           onMouseDown={this.props.handleMouseDown}
           className={visibility}>
        <h2><a href="#">Главная</a></h2>
        <h2><a href="#">О компании</a></h2>
        <h2><a href="#">Контакты</a></h2>
        <h2><a href="#">Поиск</a></h2>
      </div>
    );
  }
```

После того как вы добавили эти три строки кода, запустите приложение в своем браузере. Как только приложение будет запущено, откройте панель инструментов разработчика и посмотрите, что отображается в консоли:



Вы можете увидеть предупреждения и другие данные, но посмотрите на вывод добавленных инструкций `console.log`. Когда вы впервые запускаете свое приложение, вы увидите, что все три компонента содержат соответствующий метод `render`. Это ожидаемо, т. к. это приложение загружается в первый раз.

Не закрывая консоль, щелкните мышью по синей кнопке, чтобы открыть меню. Затем взгляните на консоль. Вы увидите новые строки (выделены полужирным шрифтом):

```
Rendering: MenuContainer
Rendering: MenuButton
Rendering: Menu
clicked
Rendering: MenuContainer
Rendering: MenuButton
Rendering: Menu
```

Когда вызывается обработчик событий `handleMouseDown`, в консоли появляется строка `clicked`. Она не важна сейчас, но наглядно разделяет серии вызовов метода `render`. С учетом вышесказанного обратите внимание, что отображение меню приводит к вызову всех трех методов `render` компонентов. Щелкните мышью, чтобы закрыть меню, и вы увидите, что все три метода `render` снова были вызваны. Так не должно быть, согласны?

Поскольку мы переключаем свойство на `Menu` и состояние хранится в компоненте `MenuContainer`, имеет смысл, чтобы методы `render` этих двух компонентов были вызваны. Но почему метод `render` компонента `MenuButton` тоже вызывается каждый раз?

После вызова метода `render` компонента `MenuContainer` вызывается компонент `Menu` и передается свойство, значение которого никогда не меняется:

```
<MenuButton handleMouseDown={this.handleMouseDown}/>
```

Значение метода `handleMouseDown` не изменяется при каждом открытии или закрытии меню. Это связано с тем, что компонент `MenuContainer` (родительский элемент компонента `MenuButton`) имеет вызванный метод `render`. Если вызывается метод `render` предка, все методы `render` дочерних компонентов также вызываются. Виной тому причина 3, в числе прочих перечисленная в начале главы, когда говорим о ситуациях, приводящих к автоматическому вызову метода `render`.

Итак, как можно предотвратить ненужный вызов метода `render` компонента `MenuButton`? Оказывается, есть два решения.

Переопределение обновления компонента

Не так давно мы рассмотрели различные методы жизненного цикла, используемые в React. Один из них — `ComponentUpdate`. Этот метод вызывается непосредственно перед вызовом метода `render`, и вы можете предотвратить вызов метода `render`, если метод

`shouldComponentUpdate` вернет значение `false`. В этом решении мы воспользуемся этой возможностью.

В код компонента `MenuButton` внесите следующие изменения (выделены цветом):

```
import React, {Component} from "react";
import "./MenuButton.css";

class MenuButton extends Component {
  shouldComponentUpdate(nextProps, nextState) {
    return false;
  }

  render() {
    console.log("Rendering: MenuButton");

    return (
      <button id="roundButton"
          onMouseDown={this.props.handleMouseDown}></button>
    );
  }
}

export default MenuButton;
```

Обновите приложение, чтобы проверить работу кода. Обратите внимание на консоль и посмотрите, что в ней будет указано, когда вы показываете и скрываете меню. Теперь вывод, когда страница загружена и меню отобразилось впервые, выглядит следующим образом:

```
Rendering: MenuContainer
Rendering: MenuButton
Rendering: Menu
clicked
Rendering: MenuContainer
Rendering: Menu
```

Обратите внимание, что метод `render` компонента `MenuButton` не был вызван. Замечательно. Тем не менее, прежде чем мы отпразднуем победу, давайте полностью решим проблему, всегда возвращая значение `false` при вызове компонента `ComponentUpdate`. Хотя этот способ сработал, нужно быть аккуратнее, чтобы не заблокировать обновления в будущем, когда будем вносить изменения в код компонента `MenuButton`.

В коде метода `shouldComponentUpdate` вы можете видеть передачу двух аргументов. Один из них — для следующего значения свойства, а другой — для следующего значения состояния. Мы можем использовать эти аргументы, чтобы сравнивать имеющееся с будущим и действовать разумнее в отношении блокировки вызова метода `render`. В случае с `MenuButton` единственное свойство, которое мы передаем, — это значение `handleMouseDown`. Мы можем проверить неизменность этого значения, внеся следующие коррективы в метод `shouldComponentUpdate`:

```
shouldComponentUpdate(nextProps, nextState) {
  if (nextProps.handleMouseDown === this.props.handleMouseDown)
  {
    return false;
  } else {
    return true;
  }
}
```

Этот код гарантирует, что мы не будем лишний раз вызывать метод `render`, если значение `handleMouseDown` остается неизменным. Если значение `handleMouseDown` изменилось, возвращается значение `true`, допуская вызов метода `render`. Вы можете использовать другие критерии, определяющие, должен ли быть вызван метод `render` компонента, все это зависит от того или иного компонента. Проявляйте творческий подход, так сказать.

Использование компонента PureComponent

Иногда возникает необходимость перекомпоновки компонентов, несмотря на отсутствие соответствующих изменений свойства или состояния. MenuButton — один из таких случаев. Решение состоит в том, чтобы вызвать метод shouldComponentUpdate и проверить, произошли ли какие-либо изменения свойства или состояния. Чтобы не делать эту проверку вручную, специальный компонент PureComponent может автоматически выполнять эту задачу.

До сих пор все наши компоненты были основаны на компоненте Component:

```
class Blah extends Component {
  render() {
    return (
      <h1>Привет!</h1>
    );
  }
}
```

Для использования компонента PureComponent, все, что нужно сделать, изменить одну строку кода:

```
class Blah extends PureComponent {
  render() {
    return (
      <h1>Привет!</h1>
    );
  }
}
```

Теперь ваш компонент будет более осторожным и станет вызывать метод render, только если определит изменение свойства или состояния. Чтобы убедиться в этом, вы можете указать зависимость MenuButton от компонента PureComponent вместо Component.

В файле *MenuButton.js* удалите метод `shouldComponentUpdate`; он больше не понадобится. Затем внесите следующие два выделенных изменения:

```
import React, {PureComponent} from "react";
import "./MenuButton.css";

class MenuButton extends PureComponent {
  render() {
    console.log("Rendering: MenuButton");

    return (
      <button id="roundButton"
          onMouseDown={this.props.handleMouseDown}></button>
    );
  }
}

export default MenuButton;
```

В первой строке мы импортируем необходимый код из библиотеки React, чтобы обеспечить поддержку компонента `PureComponent`. Затем мы расширяем компонент `MenuButton` от `PureComponent`. Если вы проверите свое приложение и взглянете на консоль после отображения меню, то увидите, что метод `render` компонента `MenuButton` при этом не вызывается (как и при сокрытии меню).

> Почему бы всегда не использовать компонент `PureComponent`?
>
> Компонент `PureComponent` кажется довольно крутым, не так ли? Почему бы нам не использовать его всегда и полностью не отказаться от `Component`? С учетом сказанного, есть несколько причин, по которым вы, возможно, захотите вернуться к `Component`.

Во-первых, `PureComponent` выполняет так называемое поверхностное сравнение. Это не полная проверка изменений свойств и состояний между вызовами для повторного рендеринга. В одних случаях это приемлемо, а в других — нет. Помните об этом при выборе в пользу компонента `PureComponent`. Может возникнуть ситуация, когда вам понадобится написать свой собственный компонент `shouldComponentUpdate` и обработать логику обновления вручную. Нельзя использовать компонент `PureComponent` и метод `shouldComponentUpdate` одновременно, хотя было бы неплохо!

Помимо логики сравнения, большая проблема с использованием `PureComponent` — это производительность. Если каждый из ваших компонентов проверяет, изменились ли свойства и состояния, даже если это поверхностная проверка, затрачивается время на вычисления. Помните, что эти проверки происходят каждый раз, когда компонент решает повторно выполнить рендеринг или просит об этом предков. Для сложных пользовательских интерфейсов такая ситуация может возникать очень часто, даже если вы не замечаете этого.

P.S: Вероятно, это должно было быть упомянуто в самом начале этого примечания: что выбрать? В принципе, лучше использовать компонент `PureComponent` вместо `Component`. Только учитывайте эти два (второстепенных) побочных эффекта.

Заключение

Обеспечение эффективности приложения требует постоянной бдительности. Анализируйте производительность своего приложения и делайте это каждый раз, когда изменяете код с целью оптимизации и повышения производительности. Каждая выполненная вами оптимизация привносит сложность, которая лишь добавляет вам (или вашей команде) дополнительные расходы на обслуживание кода и исправление багов в приложении. Будьте сознательны и не перестарайтесь. Если ваше приложение работает очень хорошо

на устройствах и в браузерах целевой аудитории (особенно малопроизводительных), считайте свою работу законченной.

> Примечание: Если у вас возникнут какие-либо проблемы, задайте вопрос!
> Если у вас есть какие-либо вопросы или код не работает ожидаемым образом, не стесняйтесь задать вопрос! Опубликуйте свой вопрос на форуме **forum.kirupa.com** и получите помощь от самых дружелюбных и знающих людей в Интернете!

Глава 18
Создание одностраничного приложения React с помощью роутера

Теперь, когда вы познакомились с основами React, давайте изучим еще кое-что. В этой главе мы применим React для создания простого **одностраничного приложения** (также называемого SPA-приложением). Как говорилось в главе 1, одностраничные приложения отличаются от традиционных многостраничных приложений, которые вы видите повсюду. Самое большое различие заключается в том, что навигация по одностраничному приложению не предполагает перехода на совершенно новую страницу. Вместо этого все страницы (обычно называемые **представлениями** в этом контексте) обычно загружаются внутри одной и той же страницы:

Когда вы загружаете контент построчно, все усложняется. Сложность не в загрузке контента. Здесь все относительно просто. Сложно сделать так, чтобы одностраничные приложения вели себя таким образом, как привыкли ваши пользователи. Уточню. Когда пользователи перемещаются по вашему приложению, у них есть некоторые ожидания:

1. URL-адрес, отображаемый в адресной строке, должен всегда отражать то, что пользователи просматривают.

2. Пользователи ожидают, что смогут успешно использовать кнопки браузера «назад» и «вперед».

3. Пользователи должны иметь возможность перейти к определенному представлению (**глубинной ссылке**) напрямую, используя соответствующий URL-адрес.

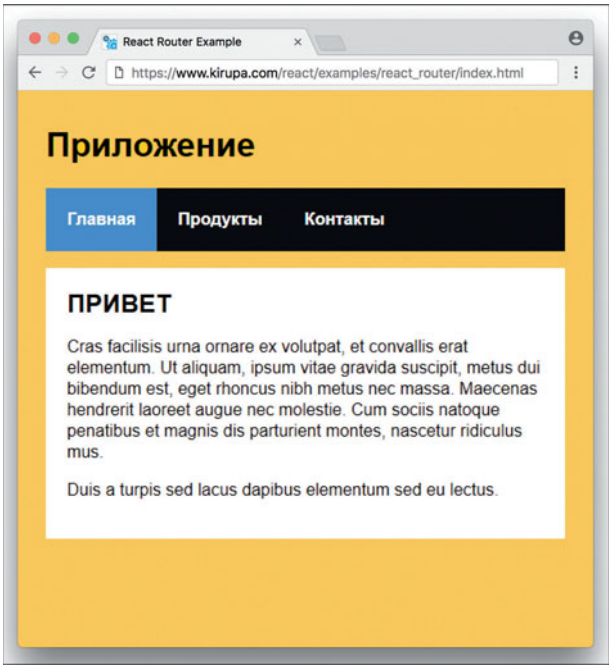
С многостраничными приложениями эти три вещи реализуемы легко. Вам даже не придется ничего делать. С одностраничным приложением, без возможности перехода на совершенно новую страницу, вам нужно, чтобы посетителю было удобно с учетом этих трех моментов, которых он ожидает от приложения. Вы должны убедиться, что за навигацию в приложении отвечают соответствующие URL-адреса. Вы должны убедиться, что история в браузере синхронизируется с каждым переходом, чтобы пользователи могли использовать кнопки «назад» и «вперед». Если пользователям нужно сделать закладку на определенное представление приложения или копировать/вставить URL-адрес для доступа к этому представлению позднее, вы должны убедиться, что ваше одностраничное приложение корректно работает и в этом направлении.

Чтобы решить все эти проблемы, существуют методы, известные как **маршрутизация**. При маршрутизации вы сопоставляете URL-адреса с обектами, которые не являются физическими страницами, такими как отдельные представления в одностраничном приложении. Это звучит сложно, но, к счастью, множество библиотек JavaScript может помочь вам в этом. Одна из таких библиотек JavaScript, самая главная, называется React Router (**github.com/reactjs/react-router**). React Router предоставляет возможности маршрутизации для одностраничных приложений, созданных в React. Что радует, так это то, что эта библиотека учитывает все, что вы уже знаете о React, и предоставляет вам удобные инструменты для работы с маршрутизацией. В этом уроке вы узнаете все о маршрутизации… и даже немножко больше.

Пример

Прежде чем читать дальше, ознакомьтесь со следующим примером в своем браузере: **www.kirupa.com/react/examples/react_router/index.html**.

Вы увидите простое приложение React, в котором используется библиотека React Router, обеспечивающая все преимущества навигации и загрузки представлений:



Переходите по пунктам меню, чтобы загрузить соответствующий раздел сайта, и используйте кнопки «назад» и «вперед», чтобы убедиться, что они работают.

В следующих разделах мы разберем это приложение на кусочки. К концу главы вы не только заново создадите это приложение, но и, надеюсь, узнаете достаточно много о React Router, чтобы создавать более функциональные и потрясающие приложения.

Начало работы

Сначала нам нужно настроить проект. Для этого мы воспользуемся старой доброй командой `create-response-app`. Перейдите к папке, в которой вы хотите создать приложение, и введите следующую команду:

```
create-react-app react_spa
```

Команда создаст в папке новый проект с именем `react_spa`. Перейдите в эту папку:

```
cd react_spa
```

Обычно мы начинаем с удаления содержимого папок, чтобы начать с чистого листа. Мы сделаем это, но сначала нужно установить React Router. Для этого выполните следующую команду:

```
npm i react-router-dom --save
```

Команда скопирует необходимые файлы React Router и зарегистрирует их в файле *package.json*, чтобы приложение было в курсе существования библиотеки React Router.

Пришло время подготовить проект, чтобы начать с нуля. В папке *react_spa* удалите все содержимое папок *public* и *src*. Теперь создадим файл *index.html*, который будет служить отправной точкой нашего приложения. В папке *public* создайте файл *index.html* и добавьте в него следующее содержимое:

```
<! DOCTYPE html>
<html>

<head>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width,
            initial-scale=1, shrink-to-fit=no">
  <title>React Router Example</title>
</head>

<body>
  <div id="root"></div>
</body>

</html>
```

Взгляните на HTML-код. Здесь нет ничего нового. Теперь мы создадим JavaScript-сценарий. В папке *src* создайте файл *index.js* и добавьте в него следующее содержимое:

```
import React from "react";
import ReactDOM from "react-dom";
import Main from "./Main";

ReactDOM.render(
  <Main/>,
  document.getElementById("root")
);
```

В коде указан вызов метода `ReactDOM.render`, а также команда рендеринга компонента `Main`… которого еще не существует. Компонент `Main` станет отправной точкой в нашей одностраничной экспедиции под ручку с React Router.

Создание одностраничного приложения

Способ создания одностраничного приложения ничем не отличается от того, как мы создавали все предыдущие приложения. У нас есть основной родительский компонент. Каждая отдельная «страница» приложения будет отдельным компонентом, который переадресовывается в главный компонент. Магическая библиотека React Router привносит в основном возможность выбора компонентов, которые нужно показать или скрыть. Вся *навигация* связана с адресной строкой браузера и кнопками «назад» и «вперед».

Отображение начального фрейма

При создании одностраничного приложения часть его страницы всегда остается статической. Эта статическая часть, также называемая **фреймом приложения**, может быть одним невидимым элементом HTML, который действует как контейнер для всего контента, или

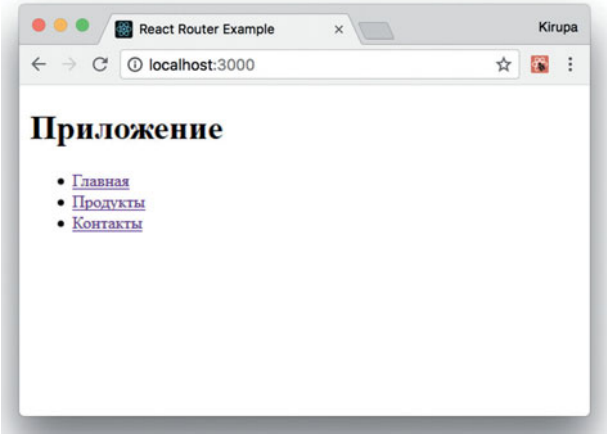
может включать в себя некоторые дополнительные видимые объекты, такие как шапка, подвал или навигация. В нашем случае фрейм приложения будет компонентом, который содержит элементы пользовательского интерфейса для шапки с навигацией и пустую область для загружаемого содержимого.

В папке *src* создайте файл *Main.js* и добавьте в него следующее содержимое:

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

class Main extends Component {
  render() {
    return (
      <div>
        <h1>Простое SPA-приложение</h1>
        <ul className="header">
          <li><a href="/">Главная</a></li>
          <li><a href="/stuff">Продукты</a></li>
          <li><a href="/contact">Контакты</a></li>
        </ul>

        <div className="content">
        </div>
      </div>
    );
  }
}

export default Main;
```

Взгляните на код. У нас есть компонент `Main`, который возвращает некоторый HTML-контент. Чтобы просмотреть результат в браузере, выполните команду `npm start`.

Вы должны увидеть неотформатированную версию названия приложения и несколько элементов списка:



Приложение не выглядит модно и стильно, но сейчас все в порядке; мы вернемся к стилям позже. Важно обратить внимание на то, что здесь нет ничего относящегося к React Router — абсолютно ничего!

Создание страниц с контентом

Наше приложение будет иметь три страницы с контентом. Этот контент будет простым компонентом, который выводит определенный JSX-код. Приступим. Сначала создайте файл с именем *Home.js* в каталоге *src* и добавьте в него следующее содержимое:

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

class Home extends Component {
  render() {
    return (
      <div>
        <h2>HELLO</h2>
        <p>Cras facilisis urna ornare ex volutpat, et
          convallis erat elementum. Ut aliquam, ipsum vitae
          gravida suscipit, metus dui bibendum est, eget
          rhoncus nibh metus nec massa. Maecenas hendrerit
          laoreet augue nec molestie. Cum sociis natoque
```

```
            penatibus et magnis dis parturient montes,
            nascetur ridiculus mus.</p>
          <p>Duis a turpis sed lacus dapibus elementum sed eu
            lectus.</p>
        </div>
      );
    }
}

export default Home;
```

Затем создайте файл *Stuff.js* в том же каталоге и добавьте в него следующий код:

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

class Stuff extends Component {
  render() {
    return (
      <div>
        <h2>Продукты</h2>
        <p>Mauris sem velit, vehicula eget sodales vitae,
        rhoncus eget sapien:</p>
        <ol>
          <li>Nulla pulvinar diam</li>
          <li>Facilisis bibendum</li>
          <li>Vestibulum vulputate</li>
          <li>Eget erat</li>
          <li>Id porttitor</li>
        </ol>
      </div>
    );
  }
}

export default Stuff;
```

У нас осталось еще одна страница. Создайте файл с именем *Contact.js* в папке *src* и добавьте в него следующий код:

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

class Contact extends Component {
  render() {
    return (
      <div>
        <h2>ЕСТЬ ВОПРОСЫ?</h2>
        <p> Опубликуйте свой вопрос на <a href="http://forum.
            kirupa.com">форуме</a>.
        </p>
      </div>
    );
  }
}

export default Contact;
```

Это все, что касается контента, который нам нужно добавить. Если вы проанализируете код, то увидите, что нет ничего проще этих компонентов. Они лишь возвращают некоторый контент с JSX-кодом. Обязательно сохраните все свои изменения в этих трех файлах. Дальше мы будем работать с ними.

Использование библиотеки React Router

У нас есть фрейм приложения в форме компонента `Main`. А также есть страницы с контентом, представленные компонентами `Home`, `Stuff` и `Contact`. Теперь нам нужно связать все это вместе, чтобы создать одностраничное приложение. И тут на сцене появляется библиотека React Router. Чтобы начать ее использовать, вернитесь к файлу *Main.js* и убедитесь, что инструкции `import` в нем выглядят следующим образом:

```
import React, {Component} from "react";
import {
  Route,
  NavLink,
  HashRouter
} from "react-router-dom";
import Home from "./Home";
import Stuff from "./Stuff";
import Contact from "./Contact";
```

Мы импортируем компоненты `Route`, `NavLink` и `HashRouter` из NPM-пакета `react-router-dom`, установленного ранее. Кроме того, мы импортируем компоненты `Home`, `Stuff` и `Contact`, потому что будем ссылаться на них как на часть загружаемого контента.

Библиотека React Router в процессе работы определяет так называемую **область маршрутизации**. В этой области находится следующее:

1. Ссылки для навигации.

2. Контейнер для загрузки контента.

Существует тесная взаимосвязь между URL-адресами навигационных ссылок и контентом, который в итоге загружается. Невозможно объяснить это на словах, не продемонстрировав на практике.

Первое, что нужно сделать, это определить область маршрутизации. В методе `render` компонента `Main` внесите следующие изменения (выделены цветом) в код:

```
class Main extends Component {
  render() {
    return (
      <HashRouter>
        <div>
          <h1>Простое SPA-приложение</h1>
          <ul className="header">
```

```
            <li><a href="/">Главная</a></li>
            <li><a href="/stuff">Продукты</a></li>
            <li><a href="/contact">Контакты</a></li>
          </ul>
          <div className="content">

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

Компонент `HashRouter` создает основу для навигации и обработки истории браузера, из которой состоит маршрутизация. Затем нам нужно определить навигационные ссылки. У нас уже есть элементы списка. Нам нужно заменить их более специализированным компонентом `NavLink`, поэтому давайте внесем следующие изменения (выделены цветом):

```
class Main extends Component {
  render() {
    return (
      <HashRouter>
        <div>
          <h1>Простое SPA-приложение</h1>
          <ul className="header">
            <li><NavLink to="/">Главная</NavLink></li>
            <li><NavLink to="/stuff">Продукты</NavLink></li>
            <li><NavLink to="/contact">Контакты</NavLink></li>
          </ul>
          <div className="content">

          </div>
        </div>
```

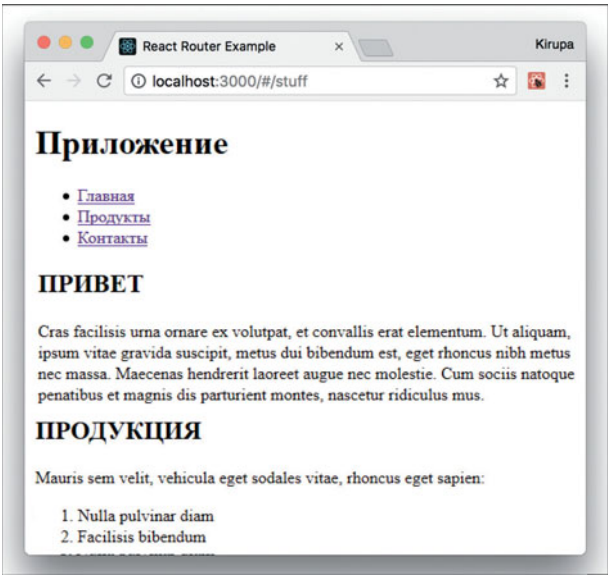
```
        </HashRouter>
      );
    }
  }
```

Обратите внимание на URL-адрес каждой ссылки, который мы сообщаем роутеру для навигации. Это значение URL-адреса (определяемое свойством) действует как идентификатор, гарантирующий загрузку правильного контента. Мы сопоставляем URL-адреса с контентом с помощью компонента `Route`. Внесите следующие изменения (выделены цветом) в код:

```
class Main extends Component {
  render() {
    return (
      <HashRouter>
        <div>
          <h1>Простое SPA-приложение</h1>
          <ul className="header">
            <li><NavLink to="/">Главная</NavLink></li>
            <li><NavLink to="/stuff">Продукты</NavLink></li>
            <li><NavLink to="/contact">Контакты</NavLink></li>
          </ul>
          <div className="content">
            <Route path="/" component={Home}/>
            <Route path="/stuff" component={Stuff}/>
            <Route path="/contact" component={Contact}/>
          </div>
        </div>
      </HashRouter>
    );
  }
}
```

Как вы видите, компонент `Route` содержит свойство `path`. Значение, присвоенное этому свойству, определяет активность пути. Когда путь активен, соответствующий компонент визуализируется. Например, когда мы переходим по ссылке `Stuff` (с путем `/stuff`, заданным свойством компонента `NavLink`), путь, чье значение свойства `path` равно `/stuff`, также становится активным. Это означает, что контент компонента `Stuff` будет визуализирован.

Вы можете увидеть все это сами. Перейдите в браузер, чтобы увидеть обновленное приложение, или выполните команду `npm start` снова. Переходите по ссылкам, чтобы увидеть, как загружается и выгружается содержимое. Содержимое главной страницы, как и ожидалось, отображается всегда, даже если мы щелкнем мышью по ссылке «Продукты» или «Контакты»:



Это проблема. Мы справимся с ней и другими небольшими неприятностями в следующем разделе, когда углубимся в изучение библиотеки React Router.

Последние штрихи

В предыдущем разделе мы создали работающее одностраничное приложение. Мы поместили всю область маршрутизации внутрь компонента `HashRouter`, и мы разделили ссылки и области, где загружаются ссылки, используя компоненты `NavLink` и `Route` соответственно. Полученный пример в целом работает, но нужно внести некоторые коррективы. В следующих разделах мы этим и займемся.

Исправление путей маршрутизации

В конце прошлого раздела мы выяснили, что в маршрутизации нашего приложения есть ошибка. Контент компонента `Home` отображается всегда, поскольку путь для загрузки компонента `Home` — `/`. Компоненты `Stuff` и `Contact` также содержат символ `/` в имени пути. Это означает, что компонент `Home` соответствует любому пути, по которому мы пытаемся перейти. Исправить это несложно. В компоненте `Route`, представляющем контент компонента `Home`, добавьте в строку пути значение `exact`, как показано ниже:

```
<div className="content">
  <Route exact path="/" component={Home}/>
  <Route path="/stuff" component={Stuff}/>
  <Route path="/contact" component={Contact}/>
</div>
```

Это свойство гарантирует, что компонент `Route` активен только в том случае, если путь к контенту точно соответствует указанному в коде. Если вы сейчас запустите свое приложение, вы увидите, что содержимое загружается правильно, при этом контент компонента `Home` отображается только тогда, когда приложение находится в представлении `Home`.

Добавление правил CSS

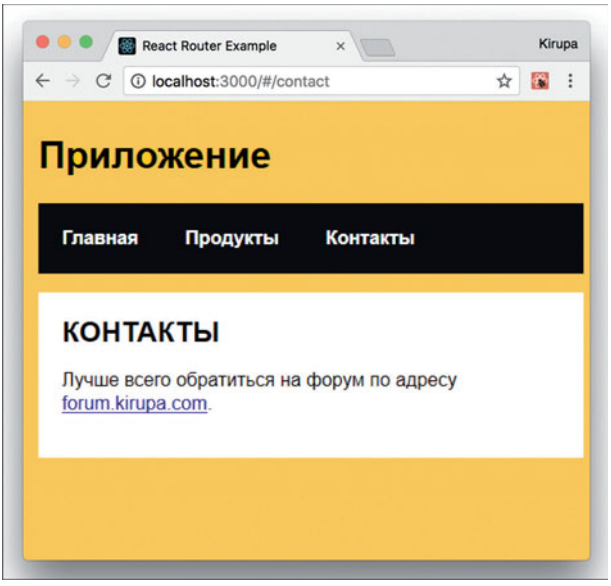
На данном этапе наше приложение еще не отформатировано. Исправить это легко. В папке *src* создайте файл *index.css* и добавьте в него следующие правила стиля:

```
body {
  background-color: #FFCC00;
  padding: 20px;
  margin: 0;
}
h1, h2, p, ul, li {
  font-family: sans-serif;
}
ul.header li {
  display: inline;
  list-style-type: none;
  margin: 0;
}
ul.header {
  background-color: #111;
  padding: 0;
}
ul.header li a {
  color: #FFF;
  font-weight: bold;
  text-decoration: none;
  padding: 20px;
  display: inline-block;
}
.content {
  background-color: #FFF;
  padding: 20px;
}
.content h2 {
  padding: 0;
  margin: 0;
}
.content li {
  margin-bottom: 10px;
}
```

Теперь нам нужно сослаться на эту таблицу стилей в коде приложения. В верхней части файла *index.js* добавьте инструкция `import`, как показано ниже:

```
import React from "react";
import ReactDOM from "react-dom";
import Main from "./Main";
import "./index.css";

ReactDOM.render(
  <Main/>,
  document.getElementById("root")
);
```

Сохраните внесенные изменения. Если вы сейчас запустите приложение, то заметите — оно уже выглядит более похожим на пример, с которого мы начинали:



Мы почти закончили! Нам нужно сделать еще пару вещей.

Выделение активной ссылки

На данном этапе в приложении трудно сказать, какая ссылка соответствует загруженному сейчас контенту. Наличие какой-то визуальной подсказки было бы полезно. Создатели библиотеки React Router продумали и это. Когда вы щелкаете мышью по ссылке, ей автоматически присваивается класс `active`.

Так, HTML-код посещенной ссылки `Stuff` выглядит так:

```
<a aria-current="true" href="#/stuff"
class="active">Продукты</a>
```

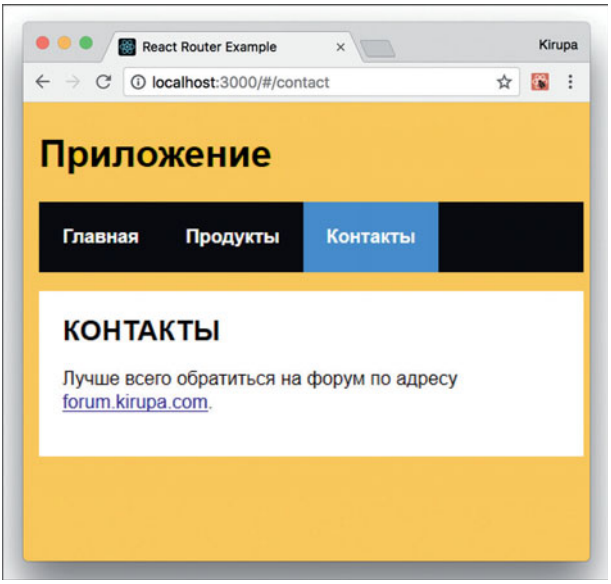
Все, что нам осталось сделать, — добавить соответствующее правило CSS, которое будет действовать, когда элементу присвоен класс `active`. Чтобы это сделать, откройте файл *index.css* и добавьте следующее правило стиля в конец документа:

```
.active {
  background-color: #0099FF;
}
```

После того как вы добавили это правило и сохранили файл, вернитесь в браузер и переходите по ссылкам в приложении. Вы увидите, что активная ссылка, содержимое которой отображается, подсвечивается синим цветом. Обратите также внимание, что ссылка «Главная» выделена всегда. Это неправильно. Исправить проблему просто: добавьте свойство `exact` в код компонента `NavLink`, представляющего контент компонента `Home`:

```
<li><NavLink exact to="/">Главная</NavLink></li>
<li><NavLink to="/stuff">Продукты</NavLink></li>
<li><NavLink to="/contact">Контакты</NavLink></li>
```

Теперь вернитесь в браузер. Вы увидите, что ссылка «Главная» подсвечивается только тогда, когда отображается содержимое главного представления:



На этом этапе мы закончили со сборкой одностраничного приложения с помощью библиотеки React Router. Ура!

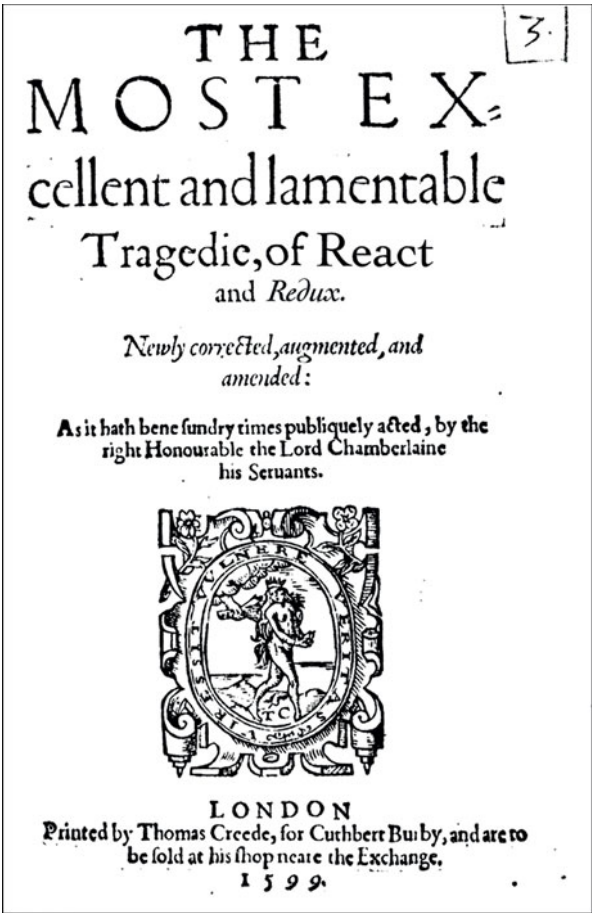
Заключение

К настоящему времени мы рассмотрели приличную часть функционала библиотеки React Router, которая помогает разрабатывать одностраничные приложения. Это не значит, что вам нечего больше изучать. Наше приложение было довольно простым, с очень скромными требованиями к маршрутизации, которую нам нужно было реализовать. React Router предоставляет намного больше (включая API-интерфейсы для всего, что вы изучили), поэтому, если вы создаете более сложное одностраничное приложение, вы должны полностью изучить документацию с примерами к библиотеке React Router (**github.com/reactjs/react-router/**).

> Примечание: Если у вас возникнут какие-либо проблемы, задайте вопрос!
> Если у вас есть какие-либо вопросы или код не работает ожидаемым образом, не стесняйтесь задать вопрос! Опубликуйте свой вопрос на форуме **forum.kirupa.com** и получите помощь от самых дружелюбных и знающих людей в Интернете!

Глава 19
Введение в Redux

Самая большая история любви всех времен и народов случилась не между Ромео и Джульеттой. И даже не между персонажами книги или кинофильма. На самом деле это произошло между React и таинственным странником из далеких земель, окрещенным именем **Redux**.


THE
MOST EX-
cellent and lamentable
Tragedie, of React
and *Redux*.

Newly corrected, augmented, and amended:

As it hath bene sundry times publiquely acted, by the right Honourable the Lord Chamberlaine his Seruants.

LONDON
Printed by Thomas Creede, for Cuthbert Burby, and are to be sold at his shop neare the Exchange.
1599.

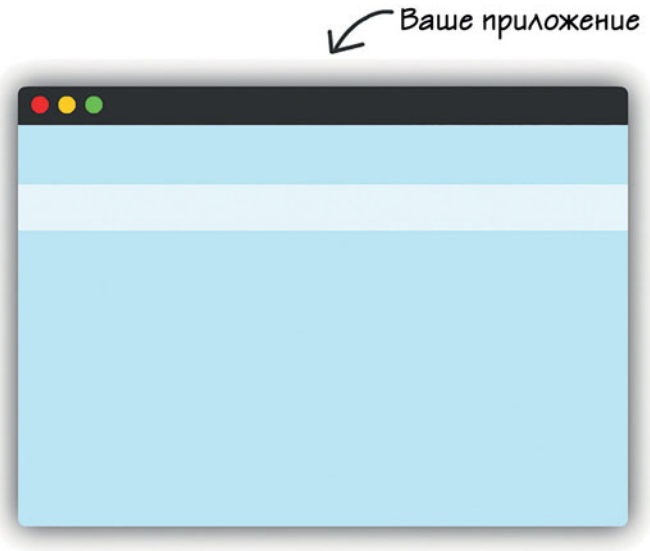
К настоящему времени вы достаточно много знаете про React, чтобы понять, как он работает и почему именно так. Однако я даже не упоминал о Redux. Нам нужно исправить ситуацию, прежде чем мы сможем понять, почему React и Redux так хорошо ладят. В следующих разделах мы изучим Redux.

Что такое Redux

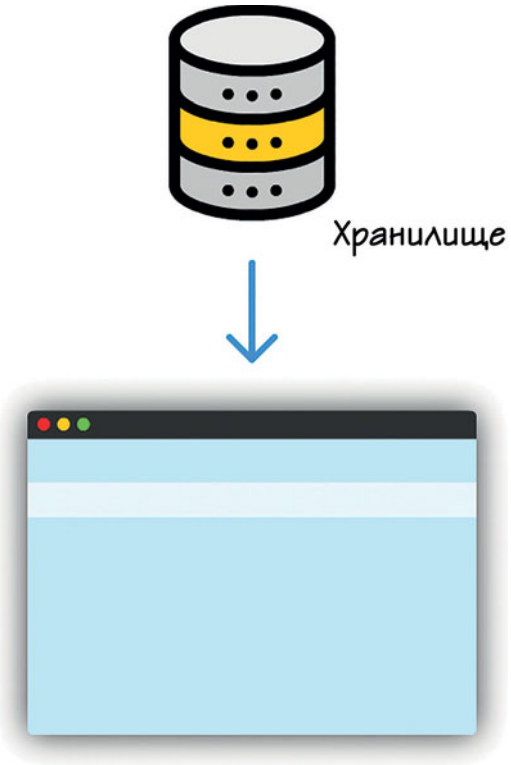
Вы поняли одну важную вещь за все это время: *следить за состоянием приложения и поддерживать его в соответствии с пользовательским интерфейсом — серьезная проблема*. Отчасти поэтому библиотеки, такие как React, стали популярны. Если вы используете более сложные проекты и заглядываете за пределы пользовательских интерфейсов, вы видите, что сохранение состояния приложения — сложное занятие в целом. Типичное приложение имеет много уровней, и каждый слой зависит от определенных данных, на которые он опирается для выполнения своих задач.

Визуализация взаимосвязи между функциональностью приложения и его состоянием часто довольно запутана:

Решить эту общую проблему поддержки состояния приложения призван **контейнер состояний Redux**. Самый простой способ понять, как работает Redux, — это изучить различные части, из которых он состоит. Первое, что нам нужно, — это приложение:

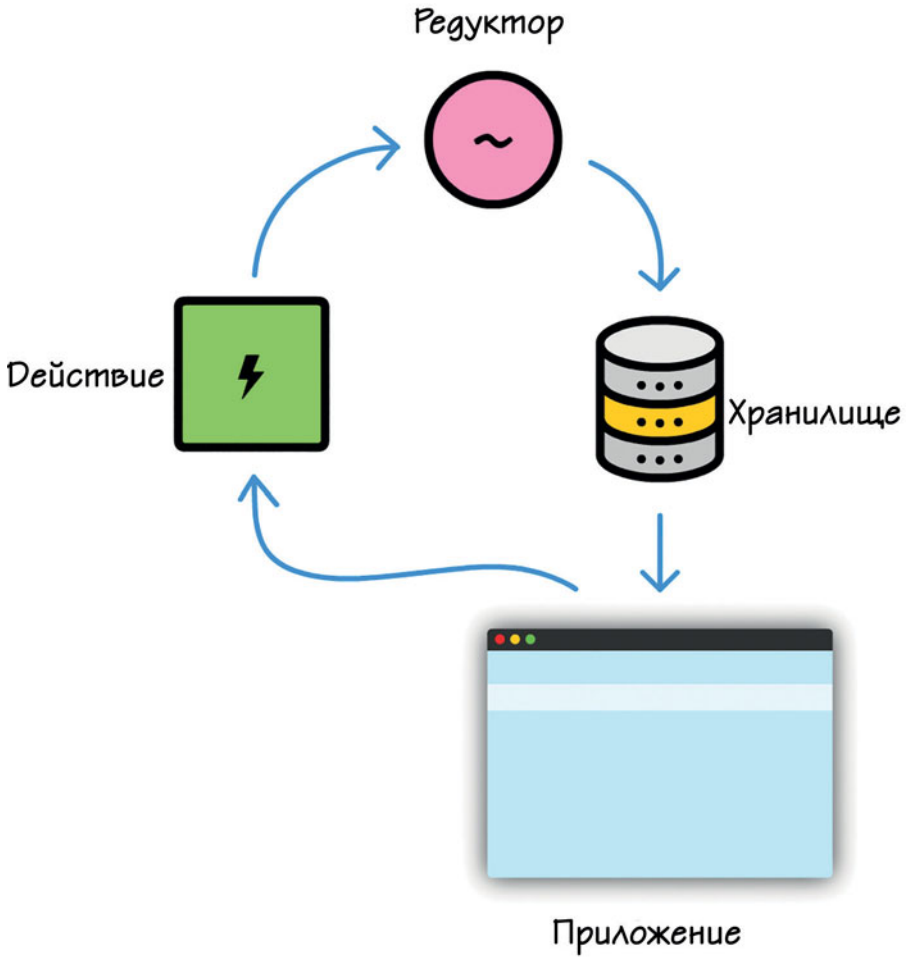


Это приложение не должно быть чем-то особенным. Оно может быть построено в React, Angular, Vue, vanilla JS или в любой другой библиотеке или фреймворке. Redux не заботится о том, *как* ваше приложение построено. Он заботится только о том, чтобы ваше приложение магическим способом работало с состоянием приложения и его сохраняло. В случае с Redux мы сохраняем **все состояния приложения** в одном месте, которое называется **хранилищем**:

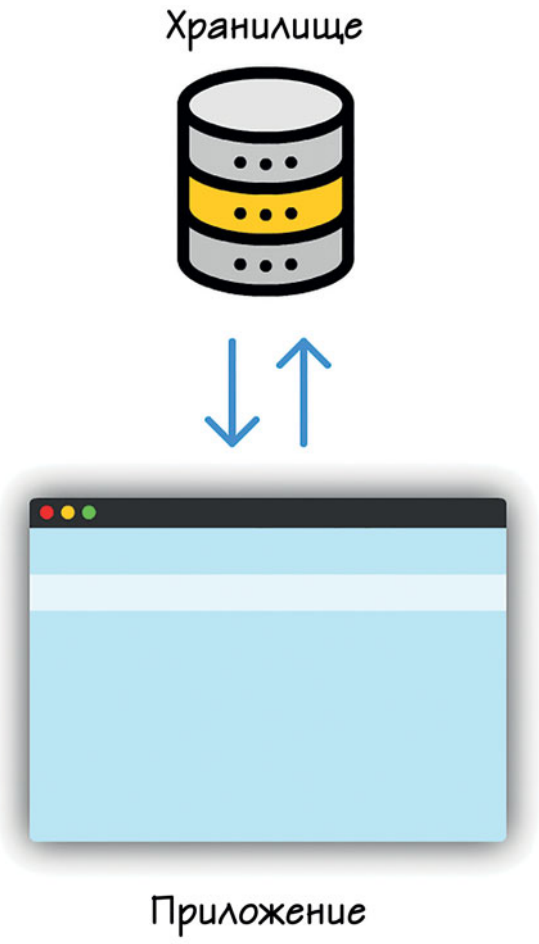


Суть хранилища заключается в том, чтобы считывать данные из него было легко. Получение информации — совсем другая история. Вы добавляете новое состояние (или изменяете текущее) в хранилище, используя комбинацию **действий**, описывающих, что нужно изменить, и **редуктор**, который определяет, какое конечное состояние будет в результате данного действия. Если вы добавите оба

изображения в предыдущий рисунок, то получится следующая картина:



На этой схеме не хватает еще нескольких частей, но это приблизительная картина происходящего, когда приложению нужно обновить состояние в хранилище. Теперь, взглянув на схему, вы, вероятно, задаетесь вопросом, почему существует все это круговое движение с окольными путями. Почему приложение не может сразу обновить хранилище?



Причина в масштабируемости. Даже для простых приложений поддержка состояния приложения в синхронизации с действиями приложения — это тяжелая работа. В сложных приложениях, в которых разные части хотят получить доступ и изменить состояние приложения, об этом не может быть и речи! Этот кольцевой способ — решение Redux для обеспечения простоты сохранения состояния приложения и для простых, и для сложных приложений. Помимо простоты, Redux помогает поддерживать состояние вашего приложения **предсказуемым**. Дэн Абрамов и Эндрю Кларк, создатели Redux, интерпретировали *предсказуемость* следующим образом:

1. **Состояние всего вашего приложения хранится в одном месте**. Вам не нужно выполнять поиск в различных хранилищах данных, чтобы найти часть своего состояния, которое вы хотите обновить. Сохранение всего в одном месте также гарантирует, что вам не нужно беспокоиться о синхронизации всех этих данных.

2. **Ваше состояние должно быть доступно только для чтения и может быть изменено только посредством действий.** Как вы видели на схеме выше, в Redux вам необходимо убедиться, что случайные части вашего приложения не смогут получить доступ к хранилищу и изменить состояние, сохраненное там. Единственный способ, которым приложение может изменить то, что находится в хранилище, — это действия.

3. **Вы указываете, каким должно быть конечное состояние.** Чтобы все было просто, состояние никогда не должно изменяться. Вы используете редуктор, чтобы указать, каким должен быть конечный результат состояния.

Эти три принципа могут показаться немного абстрактными, но когда вы начнете писать код Redux, то увидите, как они претворяются в жизнь.

Создание простого приложения с помощью Redux

Теперь мы возьмем все схемы и теорию из предыдущего раздела и попробуем писать код. Осваивая Redux, мы создадим простое консольное приложение без графического интерфейса. Это приложение будет хранить и отображать список любимых цветов. В приложении вы сможете добавлять и удалять цвета.

Вы возмутитесь, почему мы отказываемся от красивых графических приложений, которые создавали ранее? Но постойте, это приложение свяжет все теоретические знания Redux, чтобы вы поняли строки кода. Цель этой главы состоит в том, чтобы просто

понять Redux. Мы объединим Redux с пользовательским интерфейсом позже.

Настало время Redux

Сначала нам нужно создать новый HTML-документ и сослаться на библиотеку Redux. Мы не будем использовать команду `create-react-app` или любую другую чудную систему сборки. Это будет обычный простой HTML-файл, который вы можете открыть в браузере. Используя редактор кода, создайте файл с именем *favoriteColors.html* и добавьте в него следующее содержимое:

```
<! DOCTYPE html>
<html>

<head>
  <title>Избранные цвета!</title>
  <script src="https://unpkg.com/redux@latest/dist/
              redux.js"></script>
</head>

<body>
  <script>

  </script>
</body>

</html>
```

Как вы видите, у нас есть пустой HTML-документ с определенной базовой структурой. Мы ссылаемся на размещенную на сервере версию библиотеки Redux, которая отлично подходит для нашего примера. Для сложных приложений, с бóльшим функционалом, подобных созданным в React, существуют другие подходы. Мы рассмотрим их позже, но ссылку на библиотеку добавим прямо сейчас.

Свет! Камера! Мотор!

Помимо ссылки на библиотеку Redux, мы должны определить действия. Напомню, что действие — это единственный механизм, позволяющий взаимодействовать с хранилищем. В случае нашего приложения, в котором мы будем добавлять и удалять цвета, мы должны создать соответствующие действия, поддерживаемые хранилищем.

Внутри элемента `script` добавьте следующий код:

```
function addColor(value) {
  return {
    type: "ADD",
    color: value
  };
}

function removeColor(value) {
  return {
    type: "REMOVE",
    color: value
  };
}
```

У нас есть две функции: `addColor` и `removeColor`. Каждая из них принимает один аргумент и в результате возвращает объект действия. Для функции `addColor` объект действия — это две выделенные строки:

```
function addColor(value) {
  return {
    type: "ADD",
    color: value
  };
}
```

При определении действия, вам предоставляется полная свобода. Каждый объект действия имеет свойство `type`. Это ключевое слово,

которое оповещает о том, *что* вы намереваетесь делать. Кроме того, любая другая информация, которую вы отправляете вместе с действиями, полностью зависит от вас. Поскольку мы заинтересованы в добавлении или удалении значения цвета из хранилища, наш объект действия также имеет свойство `color`, которое хранит интересующий нас цвет.

Вернемся к функциям `addColor` и `removeColor`. Обе служат только одной цели — возврату действия. В мире Redux существует собственное название для таких функций. Они известны как **создатели действий**, потому что они создают действия.

Управление редуктором

Наши действия определяют то, что мы хотим сделать, а редуктор обрабатывает специфику того, что происходит и как определяется наше новое состояние. Вы можете представить редуктор как посредника между хранилищем и внешним миром, где он выполняет следующие три вещи:

1. Обеспечивает доступ к исходному состоянию хранилища;
2. Позволяет проверить действие, которое в настоящее время было запущено;
3. Позволяет установить новое состояние хранилища.

Вы можете увидеть все это на практике, добавив редуктор, чтобы вставлять и удалять цвета из хранилища. Добавьте следующий код после того, как вы определили создателей действий:

```
function favoriteColors(state, action) {
  if (state === undefined) {
    state = [];
  }
  if (action.type === "ADD") {
    return state.concat(action.color);
```

```
  } else if (action.type === "REMOVE") {
    return state.filter(function(item) {
      return item!== action.color;
    });
  } else {
    return state;
  }
}
```

Потратьте минутку, чтобы понять, как работает этот код. Вначале мы обеспечиваем то, что у нас есть некое состояние:

```
function favoriteColors(state, action) {
  if (state === undefined) {
    state = [];
  }

  if (action.type === "ADD") {
    return state.concat(action.color);
  } else if (action.type === "REMOVE") {
    return state.filter(function(item) {
      return item!== action.color;
    });
  } else {
    return state;
  }
}
```

Если объекта состояния не существует, то при первом запуске приложения мы инициализируем его как пустой массив. Вы можете использовать любую структуру данных, но массив будет самым правильным решением в нашем проекте.

Остальная часть кода отвечает за наши действия. Обратите внимание, что редуктор получает полный объект действия через свой

аргумент `action`. Это означает, что вы имеете доступ не только к свойству `type` действия, но и ко всему, что вы указали ранее как часть определения ваших действий.

В этом примере, если тип нашего действия `ADD`, мы добавляем цвет (указанный свойством `color` действия) в массив состояния. Если тип действия `REMOVE`, мы возвращаем новый массив с указанным цветом. Наконец, если тип действия — то, чего мы не знаем, мы возвращаем текущее состояние, немодифицированное:

```
function favoriteColors(state, action) {
  if (state === undefined) {
    state = [];
  }

  if (action.type === "ADD") {
    return state.concat(action.color);
  } else if (action.type === "REMOVE") {
    return state.filter(function(item) {
      return item !== action.color;
    });
  } else {
    return state;
  }
}
```

Довольно просто, не так ли? Обязательно учтите следующие моменты при работе с редукторами в Redux. В документации Redux они описаны лучше всего (**redux.js.org/docs/basics/Reducers.html**).

Вещи, которые вы *никогда* не должны делать внутри редуктора:

- Изменять свои аргументы.
- Выполнять сторонние действия, такие как вызовы API и переходы маршрутизации.

- Вызывать не чистые функции, например, `Date.now()` или `Math.random()`.

Учитывая те же аргументы, он должен вычислить следующее состояние и вернуть его. Без сюрпризов. Никаких сторонних действий. Никаких вызовов API. Никаких изменений. Только вычисления.

Вы можете увидеть все это в коде нашего примера. Чтобы добавить новые значения цвета в массив состояний, мы использовали метод `concat`: он возвращает совершенно новый массив, состоящий из старых значений и нового значения, которое мы добавили. Функция `push` даст нам тот же результат, но это нарушает нашу задачу, заключающуюся в отсутствии изменений в существующем состоянии. Чтобы удалить значения цвета, мы продолжаем придерживаться нашей цели — не изменять текущее состояние. Мы используем метод `filter`, который возвращает новый массив со значением, которое мы хотим удалить.

Также Марк Эриксон (**@acemarke**) напомнил мне: *Redux не содержит никакой механики, чтобы мы не могли изменять состояние и совершать другие плохие вещи.* Создатели Redux предоставили некоторые рекомендации. Мы должны следовать им и применять эти рекомендации на практике.

Работа с хранилищем

Все, что осталось сейчас сделать, это связать действия и редуктор с хранилищем. Сначала мы должны создать хранилище. Сразу после функции `favoriteColor` добавьте следующий код:

```
var store = Redux.createStore(favoriteColors);
```

Здесь мы создаем новое хранилище, используя метод `createStore`. Аргумент, который мы приводим, — это `favoriteColor` редуктор, который мы создали несколько мгновений назад. Теперь у нас есть полный цикл Redux для хранения состояния приложения. У нас есть хранилище, редуктор и действия, которые сообщают редуктору, что делать.

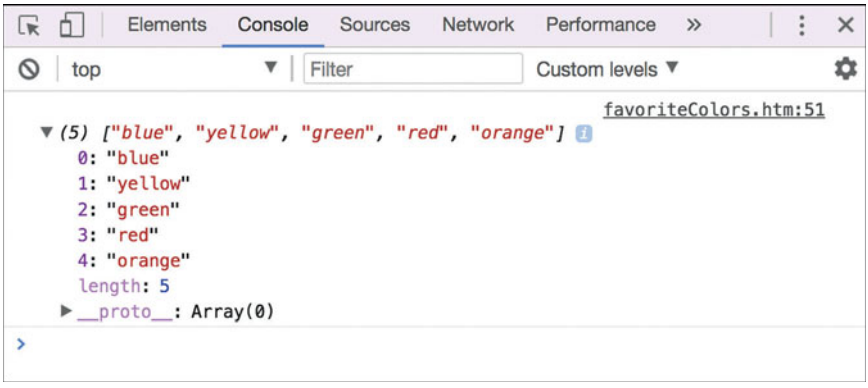
Чтобы посмотреть, как все это работает, мы добавим (и удалим) некоторые цвета в (из) хранилище. Для этого мы воспользуемся методом `dispath` на объекте `store`, который принимает действие в качестве аргумента. Переходим дальше и добавим следующие строки:

```
store.dispatch(addColor("blue"));
store.dispatch(addColor("yellow"));
store.dispatch(addColor("green"));
store.dispatch(addColor("red"));
store.dispatch(addColor("gray"));
store.dispatch(addColor("orange"));
store.dispatch(removeColor("gray"));
```

Каждый вызов `dispath` посылает действие реуктору. Последний принимает меры и выполняет соответствующую работу для определения нового состояния. Чтобы увидеть текущее состояние хранилища, вы можете добавить следующую инструкцию после всех вызовов `dispath`:

```
console.log(store.getState());
```

Как следует из названия, метод `getState` возвращает значение состояния. Если вы просматриваете свое приложение в браузере и используете инструменты разработчика, вы увидите, что добавленные нами цвета отображаются в консоли:



Мы почти закончили. У нас есть еще одна действительно важная вещь, которую нужно учесть. В реальных сценариях вы хотите получать уведомления каждый раз, когда изменяется состояние приложения. Эта модель уведомлений сделает жизнь намного проще, если вы хотите обновить интерфейс или выполнить другие задачи в результате некоторых изменений в хранилище. Для этого существует метод `subscribe` для указания функции, которая вызывается каждый раз, когда содержимое хранилища изменяется. Чтобы увидеть метод `subscribe` в действии, сразу после того, как вы определили объект хранилища, внесите следующие изменения (выделены цветом):

```
var store = Redux.createStore(favoriteColors);
store.subscribe(render);

function render() {
  console.log(store.getState());
}
```

После того, как вы это сделаете, снова запустите приложение. На этот раз, когда вы вызываете `dispath` для запуска другого действия, функция `render` вызывается при изменении хранилища.

Заключение

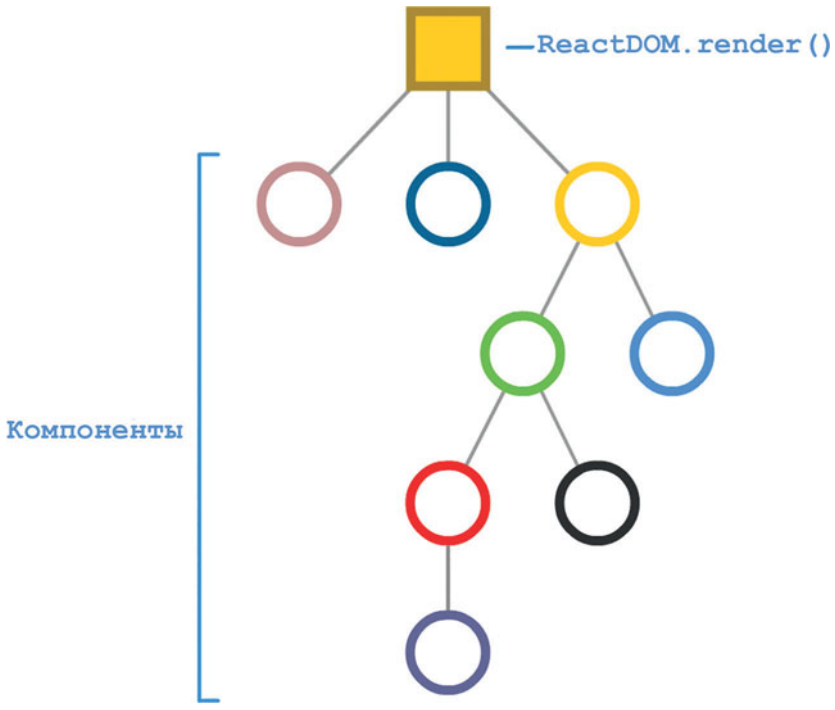
Мы совершили ознакомительный тур по библиотеке Redux и основным функциональным возможностям, которые она привнесла. Мы рассмотрели случаи, в которых Redux действительно полезна для работы с состоянием приложения, а также попрактиковались с кодом. Единственное, чего нам не удалось сделать, это создать более реалистичный пример приложения. Redux — достаточно гибкая библиотека, чтобы работать с любой инфраструктурой пользовательского интерфейса, и у каждого пользовательского интерфейса есть своя магия при работе с Redux. Разумеется, наш выбор пользовательского интерфейса — React! Мы рассмотрим, как связать их вместе, в следующей главе.

Примечание: Если у вас возникнут какие-либо проблемы, задайте вопрос!

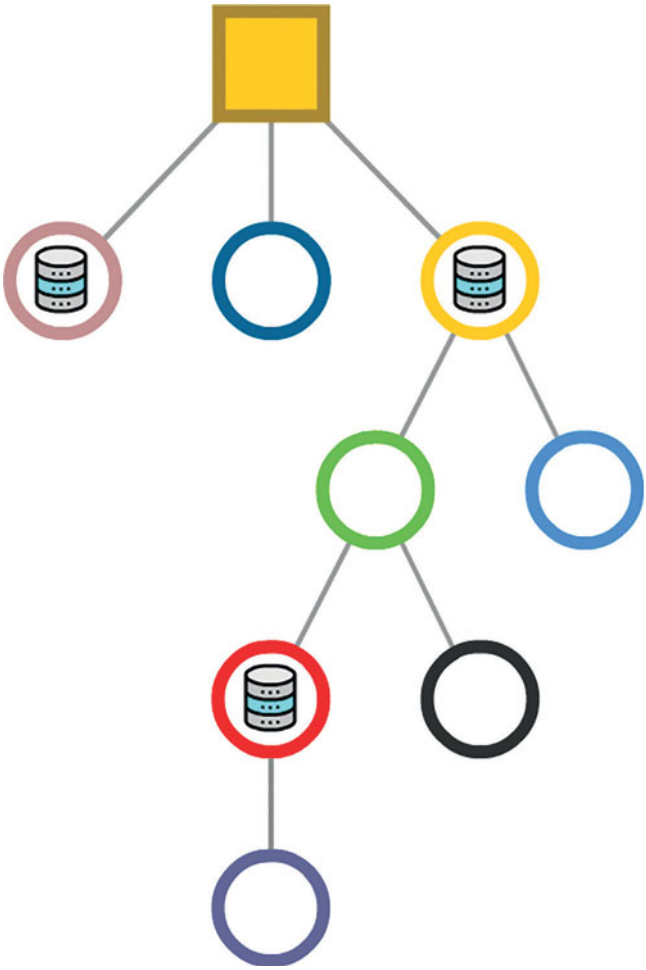
Если у вас есть какие-либо вопросы или код не работает ожидаемым образом, не стесняйтесь задать вопрос! Опубликуйте свой вопрос на форуме **forum.kirupa.com** и получите помощь от самых дружелюбных и знающих людей в Интернете!

Глава 20
Совместное использование Redux с React

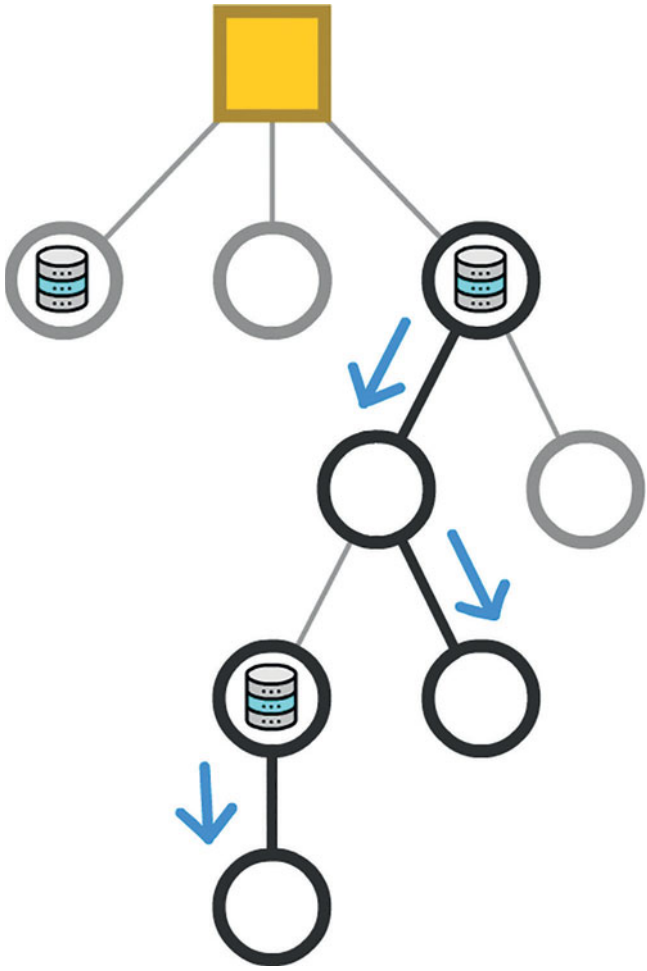
Теперь, когда вы выяснили, как работает Redux, давайте рассмотрим тему, до которой мы добрались, чтобы лучше понять: *почему Redux настолько популярен в проектах React?* Для ответа взгляните на следующую иерархию компонентов любого произвольного приложения:



Единственная деталь, которую мы будем использовать, такова: некоторые из этих компонентов отвечают за управление состоянием и передачу этого состояния в виде свойств:

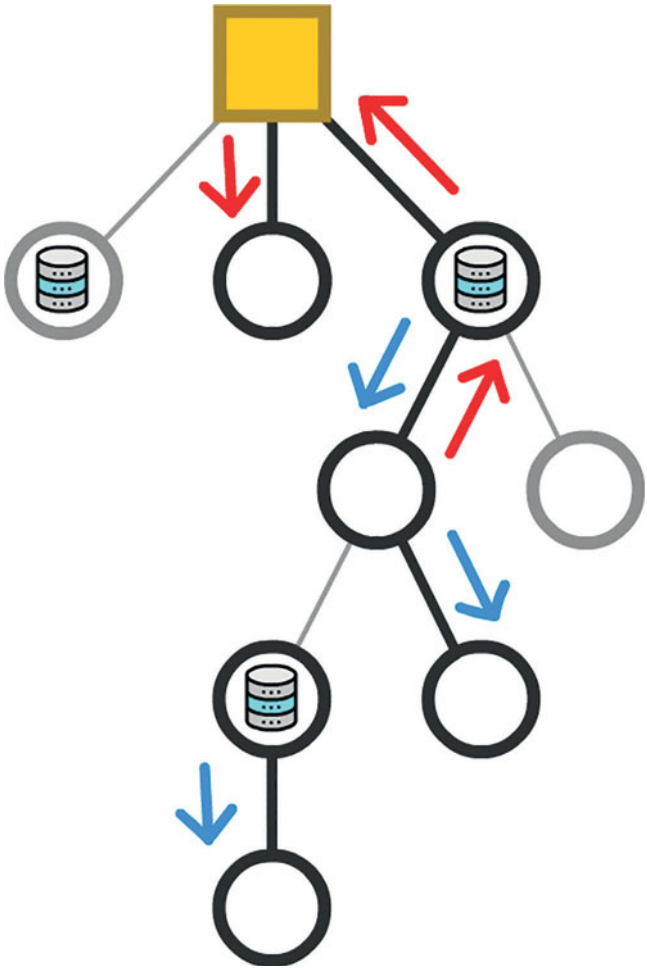


В идеальном проекте данные, необходимые каждому компоненту, аккуратно переносятся от предка к потомку:



К сожалению, за пределами простых сценариев то, что мы хотим сделать, не всегда достижимо. Типичное приложение содержит много состояний, генерирующих, обрабатывающих и передающих. Один компонент может инициировать изменение состояния. Другой компонент в другом месте захочет отреагировать на него.

Свойства, связанные с этим изменением состояния, могут перемещаться как вниз по дереву, так и вверх, чтобы охватить любой компонент, полагающийся на передаваемые данные:



Мы были замешаны в этом довольно часто, когда передавали что-либо (значение переменной, ссылка на функцию/обработчик событий и т. д.) от потомка к предку и за его пределами.

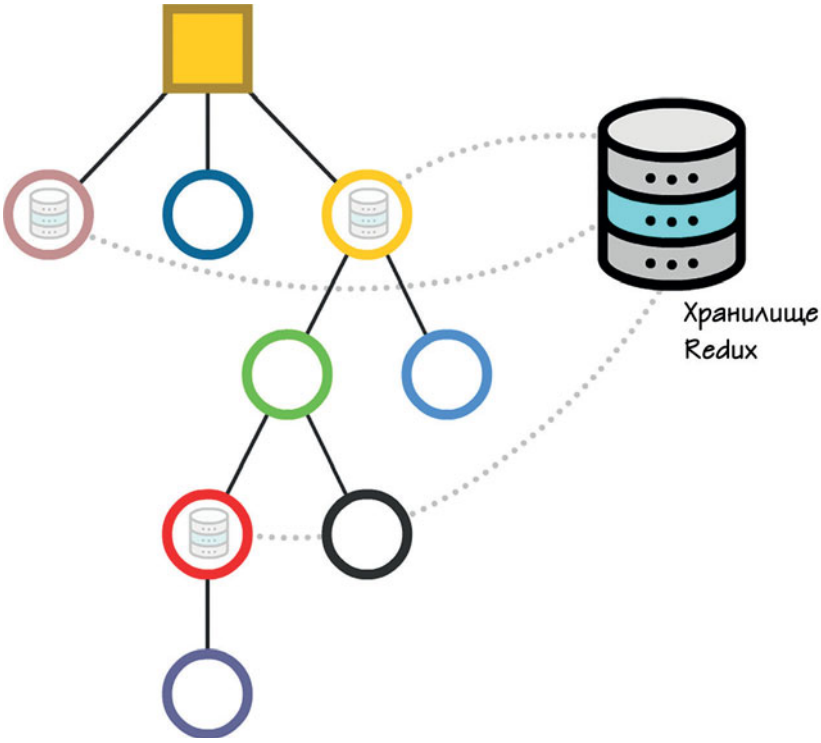
На этом этапе нам нужно признать несколько проблем, которые могут возникнуть в результате передачи данных по нашим компонентам:

1. **Зависимости усложняют обслуживание кода.** Задача React заключалась в том, чтобы избежать спагеттиподобных зависимостей. Когда у нас есть данные, окружающие наше приложение, мы получаем именно то, от чего должны были освободиться.

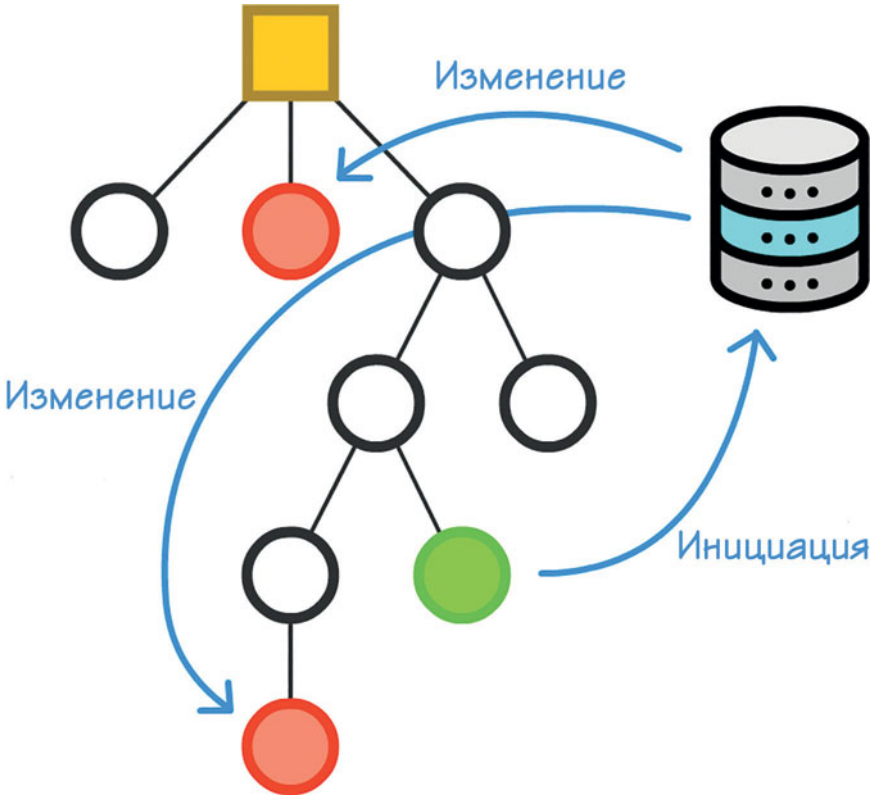
2. **Каждый раз, когда состояние изменяется или передается свойство, все затронутые компоненты запрашиваются на повторный рендеринг.** Чтобы пользовательский интерфейс был синхронизирован с текущим состоянием, это поведение довольно неплохое. Как уже упоминалось, многие компоненты не требуют повторного рендеринга, когда они просто передают значение от предка к потомку, без дополнительного ввода. Мы рассмотрели способы минимизации этого повторного рендеринга, настройкой `mustComponentUpdate` или полагаясь на компонент `PureComponent`, но оба подхода — сложны в синхронизации данных, т. к. потребности приложения должны развиваться.

3. **Наша иерархия компонентов имитирует пользовательский интерфейс, а не данные.** То, как мы организуем и размещаем компоненты, помогает разделить интерфейс на более мелкие и управляемые части. Это правильный подход. Несмотря на правильность, компоненты, которые инициируют изменение состояния, и те, которые должны реагировать на него, часто не находятся в одном и том же родственном отношении (предок/потомок/сестра). Это требует, чтобы свойства передавались на большие расстояния, часто несколько раз за каждое изменение.

Решение этих проблем — это Redux. Redux не полностью решает их, но с большей частью справляется. Redux позволяет вам управлять состоянием приложения внутри своего хранилища данных, а не распределяться через кучу компонентов:

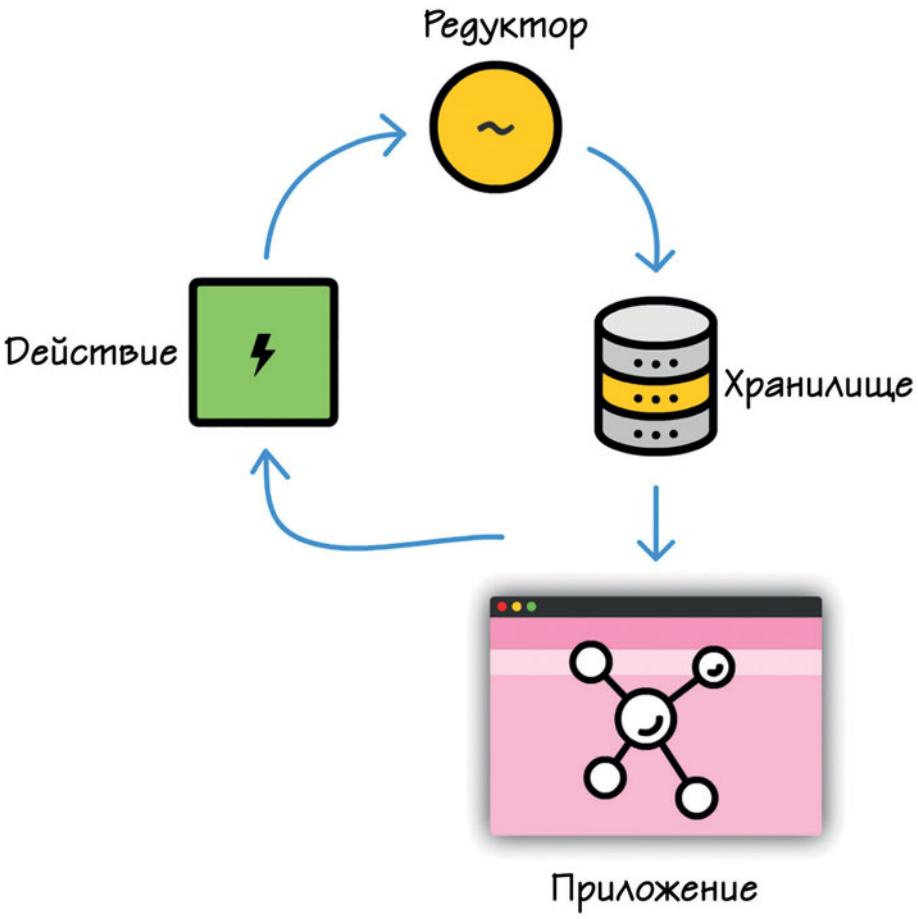


Этот подход решает несколько проблем. Если вы хотите поделиться данными из одной части приложения с другой, вы можете сделать это, не перемещаясь вверх и вниз по иерархии компонентов:



Вы можете инициировать изменение состояния и вовлечь только те компоненты, на которые воздействуют напрямую. Это уменьшает количество расходов, которые вам в противном случае пришлось бы поддерживать, чтобы данные (и любые изменения) попадали в место назначения, не вызывая ненужные методы `render`. Довольно круто, не так ли?

Теперь давайте перейдем на один уровень выше. С архитектурной точки зрения обзор Redux, который вы получили в прошлой главе, показан ниже:



Помимо хранилища, нам все равно придется работать с действиями, редуктором и всеми другими связанными с ними объектами, которые составляют Redux. Единственное отличие состоит в том, что наше приложение построено с использованием React, и на этой разнице (и на том, как происходит взаимодействие с Redux) мы и сфокусируем наше внимание.

Управление состояниями React с помощью Redux

То, как Redux подключается к вашему React-приложению, так же просто, как вызов нескольких API-интерфейсов Redux из кода React. Доступно два шага:

Предоставьте приложению ссылку на Redux.

Сопоставьте создателей действий, функции отправки и состояния в качестве свойств для любого компонента, который нуждается в данных из хранилища.

Чтобы разобраться с реализацией этих двух шагов, мы создадим простое приложение-счетчик, которое показано на рис. 20.1:

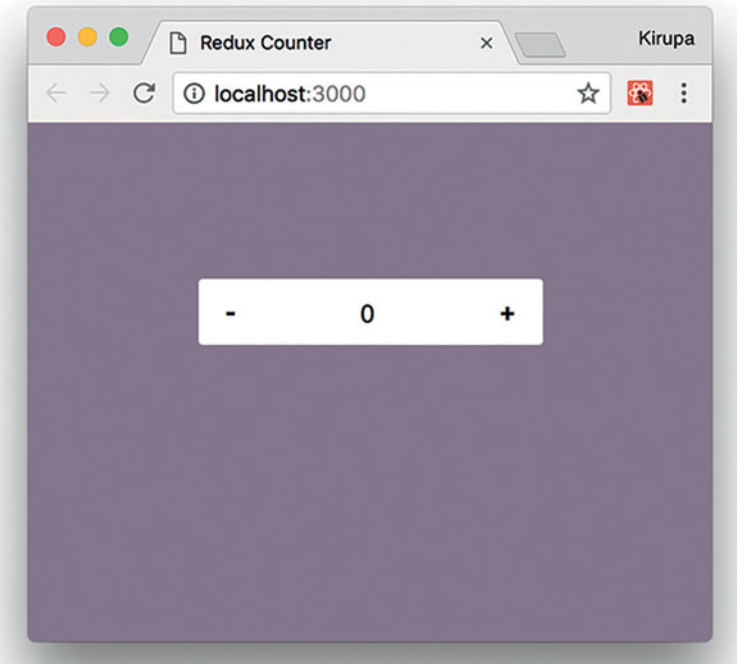


Рис. 20.1. Пример приложения-счетчика, которое мы создадим

Наше приложение будет иметь кнопки «плюс» и «минус», чтобы увеличивать и уменьшать значение счетчика. Сейчас ничего не происходит; это всего лишь подходящий уровень функциональности и сложности, который поможет вам попрактиковаться, объединив React и Redux.

Как пересекаются React и Redux

Обычно мы начинаем копировать и вставлять HTML, CSS и JavaScript-код, чтобы запустить демонстрационное приложение. Через несколько мгновений мы к этому и перейдем, но сначала нам нужно понять структуру приложения. Без учета данных и управления состоянием у нас будет только два компонента (рис. 20.2):

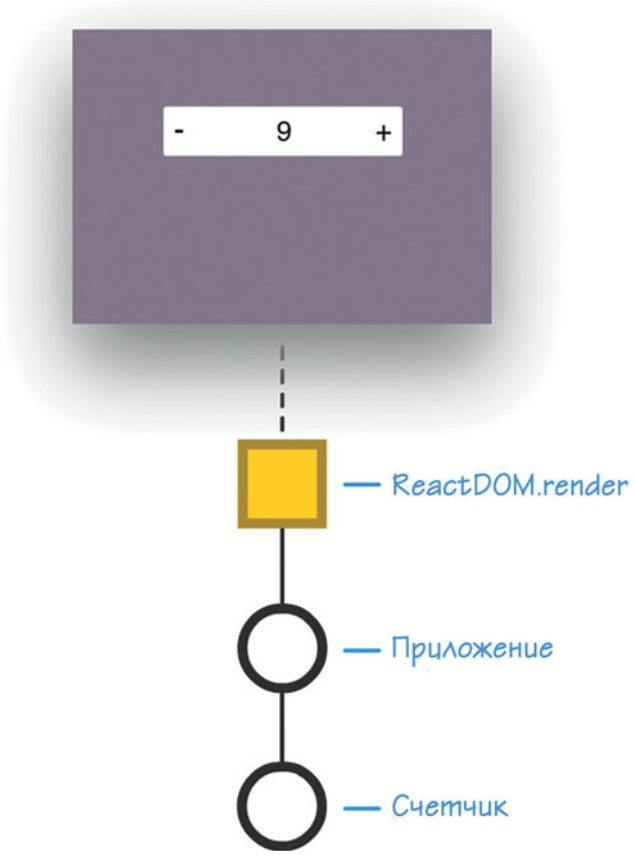


Рис. 20.2. Как устроено наше приложение

У нас будут компоненты `App` и `Counter`. Теперь счетчик кажется не самым сложным примером приложения, верно? Реализуя его с использованием обычного состояния, мы бы создали объект состояния в компоненте `Counter` и ввели переменную, значение которой

увеличивается или уменьшается в зависимости от того, какую кнопку мы нажимаем.

Когда проект смешивает это с Redux, схема компонентов становится немного странной. Она будет выглядеть так, как показано на рис. 20.3:

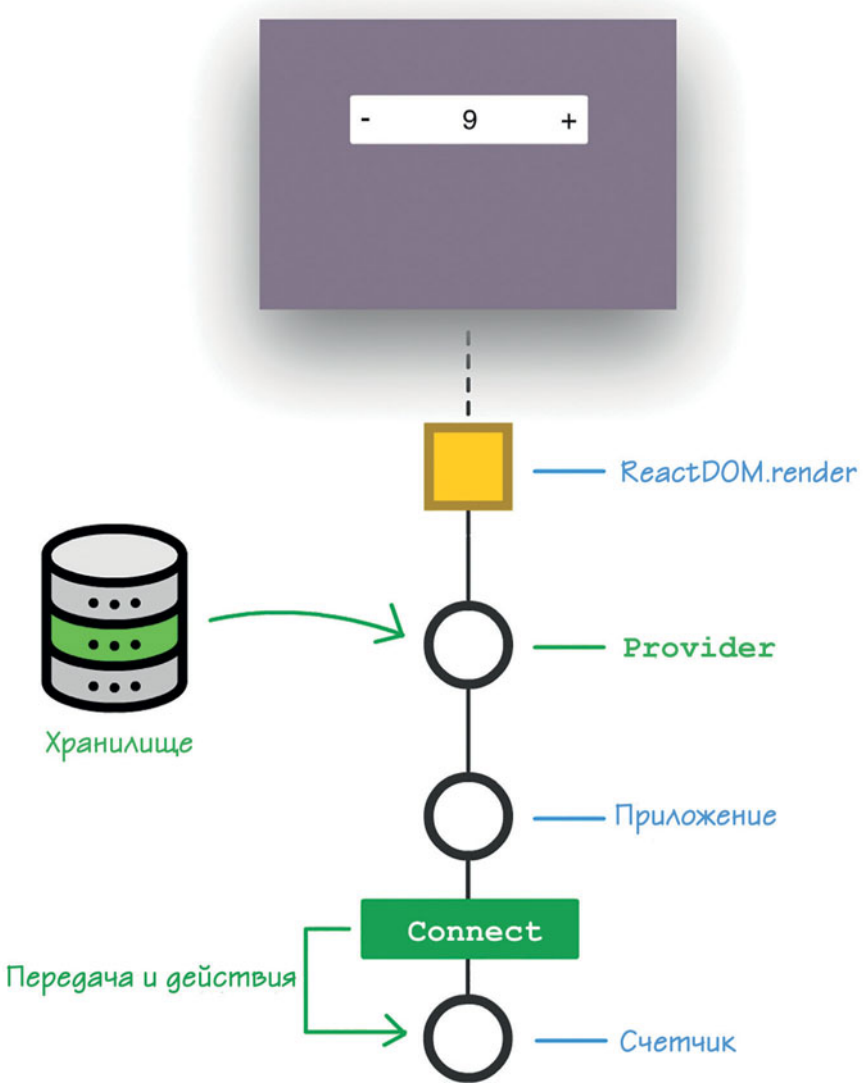


Рис. 20.3. Структура приложения после добавления Redux

Элементы синего цвета мы имели изначально. Элементы зеленого цвета — новые, т. к. мы добавили Redux в наше приложение. Ранее упоминалось, что добавление Redux в приложение включает два шага. Зеленые дополнения имитируют эти шаги:

Первый шаг для обеспечения доступа к хранилищу Redux выполняется компонентом `Provider`.

Второй шаг, предоставление любому заинтересованному компоненту доступа к нашим действиям, — выполняется компонентом `Connect`.

Если углубляться в детали, компонент `Provider` — шлюз для получения функциональности Redux в нашем приложении React. Он несет ответственность за хранение ссылки на хранилище и обеспечение того, чтобы все компоненты в приложении имели доступ к нему. Он может сделать это, будучи самым верхним компонентом в иерархии компонентов. Эта позиция позволяет ему легко передавать все данные, связанные с Redux, по всему приложению.

Компонент `Connect` немного интереснее. Это не полномасштабный компонент в традиционном смысле. Он известен как Higher Order Component (компонент более высокого порядка — **reactjs.org/docs/higherorder-components.html**), или HOC, как говорят крутые программисты. Такие компоненты обеспечивают последовательный способ расширить функциональность существующего компонента, расширяя его и вводя в него свои дополнительные функциональные возможности. Представьте как способ в духе React имитировать функционал ключевого слова `extends` при работе с ES6-классами. Схема показывает нам: конечный результат заключается в том, что благодаря компоненту `Connect` компонент `Counter` имеет доступ к любым действиям и передаче, необходимым для работы с хранилищем Redux без необходимости писать какой-либо специальный код для его доступа. Компонент `Connect` позаботится об этом.

Компоненты более высокого порядка, как `Provider`, так и `Connect`, создают отношения, которые позволяют любому обычному React-приложению легко работать с уникальным (но полностью эффективным и удивительным) режимом управления состоянием

приложения Redux. Когда мы начнем создавать приложение, вы увидите, как это происходит, на практике.

Начало работы

Теперь, когда у вас есть представление о том, как наше приложение будет структурировано, а также о некоторых конструктивных элементах Redux, которые мы будем использовать, давайте создадим приложение. Чтобы начать работу, сначала выполните команду `create-react-app` для создания приложения, которое мы назовем `reduxcounter`:

```
create-react-app to create an app we'll call reduxcounter:
create-react-app reduxcounter
```

Теперь установим зависимости Redux и React Redux. В оболочке командной строки перейдите к папке *reduxcounter* и выполните следующую команду:

```
npm install redux
```

Эта команда устанавливает библиотеку Redux так, что наше приложение может использовать базовые инструменты Redux, которые предоставляют возможность работать с состоянием приложения. После того, как библиотека Redux полностью установлена, нам нужно разобраться с еще одной зависимостью. Выполните следующую команду, чтобы передать все содержимое React Redux:

```
npm install react-redux
```

Когда эта команда завершится, у нас будет все необходимое, чтобы собрать приложение React и использовать в нем некоторую магию Redux. Пришло время начать!

Создание приложения

Сначала нам нужно очистить проект от ненужных файлов. Перейдите в папки *src* и *public* и удалите все их содержимое. Затем в папке *public* создайте файл *index.html* и добавьте в него следующий HTML-код:

```
<! DOCTYPE html>
<html>

<head>
  <title>Счетчик Redux</title>
</head>

<body>
  <div id="container">

  </div>
</body>

</html>
```

Единственное, что нужно отметить, это то, что у нас есть элемент `div` с идентификатором `container`.

Затем создадим JavaScript-сценарий, который станет точкой входа в наше приложение. В папке *src* создайте файл *index.js* и добавьте в него следующее содержимое:

```
import React, {Component} from "react";
import ReactDOM from "react-dom";
import {createStore} from "redux";
import {Provider} from "react-redux";
import counter from "./reducer";
import App from "./App";
import "./index.css";
```

```
var destination = document.querySelector("#container");

// Хранилище
var store = createStore(counter);
ReactDOM.render(
  <Provider store={store}>
    <App />
  </Provider>,
  destination
);
```

Проанализируйте код. Сначала мы инициализируем хранилище Redux и используем надежный метод `createStore`, который принимает редуктор в своем аргументе. Редуктор ссылается на переменную `counter`, и, если вы посмотрите на инструкции `import`, вы увидите, что они определяются в файле с именем *reducer.js*. Мы скоро создадим его.

После создания хранилища мы предоставляем его в качестве свойств для компонента `Provider`. Компонент `Provider` предназначен для использования в качестве внешнего компонента в приложении, чтобы гарантировать, что каждый компонент имеет доступ к хранилищу `Redux` и связанной с ним функциональности:

```
ReactDOM.render(
  <Provider store={store}>
    <App />
  </Provider>,
  destination
);
```

Далее создадим редуктор. Вы уже видели, что редуктор ссылается на переменную `counter` и находится внутри файла с именем *reducer.js*, которого не существует. Исправьте это, сначала создав

файл с именем *reducer.js* в папке. Создав его, добавьте в него следующий JavaScript-код:

```
// Редуктор
function counter(state, action) {
  if (state === undefined) {
    return {count: 0};
  }

  var count = state.count;

  switch (action.type) {
    case "increase":
    return {count: count + 1};
    case "decrease":
    return {count: count – 1};
    default:
    return state;
  }
}

export default counter;
```

Наш редуктор довольно прост. У нас есть переменная `count`, которой мы присваиваем значение `0`, если состояние пустое. Этот редуктор будет иметь дело с двумя типами действий: `increase` и `decrease`. Если используется действие `increase`, мы увеличиваем значение переменной `count` на `1`. Если используется действие `decrease`, то уменьшаем значение переменной `count` на 1.

На данный момент мы примерно на полпути (рис. 20.4):

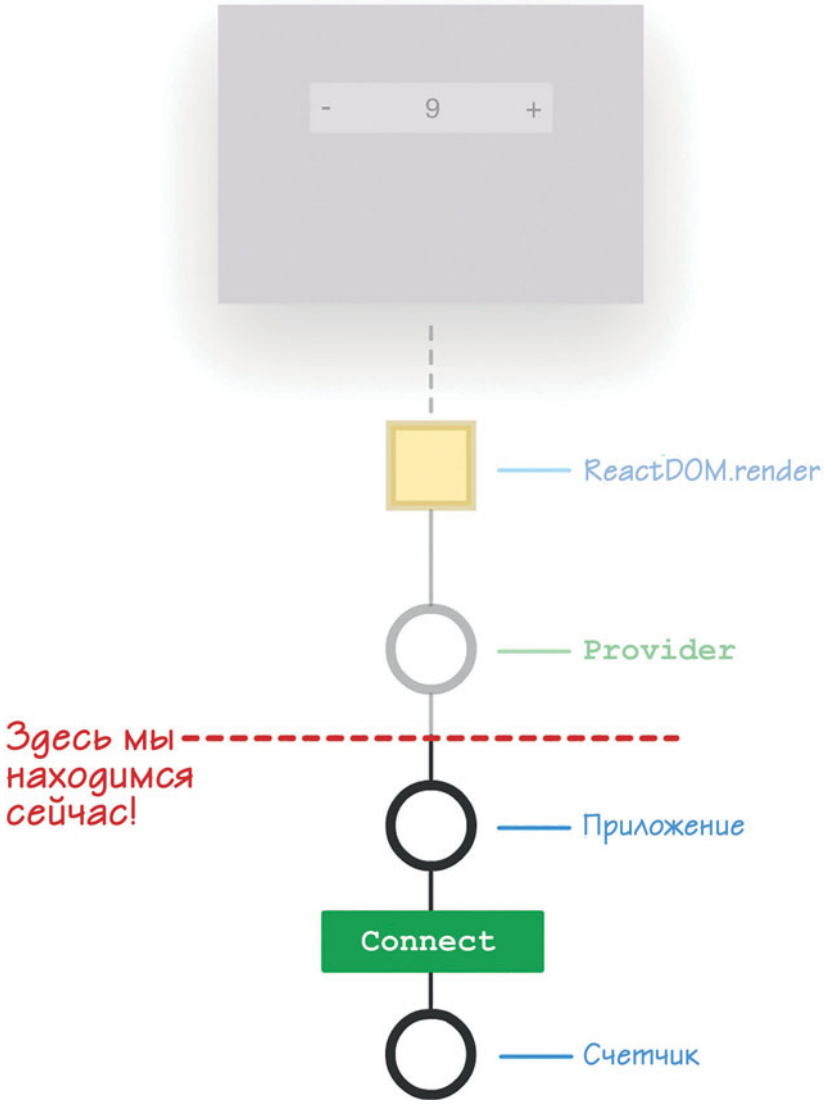


Рис. 20.4. Здесь мы находимся сейчас!

Мы должны опуститься на один уровень вниз в приложении и заняться компонентом `App`. В папке *src* создайте файл под названием *App.js*. В этот файл добавьте следующее:

```
import {connect} from "react-redux";
import Counter from "./Counter";

// Сопоставление состояния Redux со свойствами компонента
function mapStateToProps(state) {
  return {
    countValue: state.count;
  };
}

// Действие
var increaseAction = {type: "increase"};
var decreaseAction = {type: "decrease"};

// Сопоставление действий Redux со свойствами компонента
function mapDispatchToProps(dispatch) {
  return {
    increaseCount: function() {
      return dispatch(increaseAction);
    },
    decreaseCount: function() {
      return dispatch(decreaseAction);
    }
  };
}

// HOC-компонент
var connectedComponent = connect(
  mapStateToProps,
  mapDispatchToProps
)(Counter);

export default connectedComponent;
```

Потратьте несколько минут, чтобы изучить код. Основная цель этого кода в том, чтобы превратить все, относящееся к Redux, в то, что можно использовать в React. Другими словами, мы предоставляем специальные свойства, которые компонент может легко использовать с помощью двух функций, называемых `mapStateToProps` и `mapDispatchToProps`.

Прежде всего функция `mapStateToProps`:

```
// Сопоставление состояния Redux со свойствами компонента
function mapStateToProps(state) {
  return {
    countValue: state.count;
  };
}
```

Эта функция подписывается на все обновления хранилища и вызывается, когда что-либо в хранилище изменяется. Она возвращает объект, содержащий данные хранилища, которые вы хотите передать в качестве свойств компоненту. В нашем случае то, что мы передаем, довольно просто: объект, который содержит свойство `countValue`, значение которого представлено свойством `count` из хранилища.

Предоставление значения хранилища в качестве свойств — это только первая часть того, что нам нужно сделать. Затем нужно предоставить нашему компоненту доступ к действиям в виде свойств. Следующий код отвечает за это:

```
// Действие
var increaseAction = {type: "increase"};
var decreaseAction = {type: "decrease"};

// Сопоставление действий Redux со свойствами компонента
function mapDispatchToProps(dispatch) {
  return {
    increaseCount: function() {
      return dispatch(increaseAction);
```

```
    },
    decreaseCount: function() {
      return dispatch(decreaseAction);
    }
  };
}
```

Действительно интересное происходит с функцией `mapDispatchToProps`. Мы возвращаем объект, содержащий имена двух функций, которые компонент может вызвать для отправки изменений в хранилище. Функция `increaseCount` запускает отправку с типом действия `increase`. Функция decreaseCount запускает отправку с типом действия `decrease`. Если вы посмотрите на редуктор, который мы добавили несколько минут назад, вы увидите, как любой из этих вызовов функций повлияет на значение `count`, которое мы храним в хранилище.

Все, что осталось, — это гарантировать, что любой компонент, который мы хотим предоставить всем этим свойствам, имеет некий способ их получения. Вот где вступает в силу волшебная функция `connect`:

```
var connectedComponent = connect(
  mapStateToProps,
  mapDispatchToProps
)(Counter);
```

Эта функция создает магический HOC-компонент `Connect`, о котором мы говорили ранее. Он принимает команды `mapStateToProps` и `mapDispatchToProps` как аргументы, и передает все это в указанный компонент `Counter`. Конечный результат всего этого кода эквивалентен визуализации следующего:

```
<Connect>
  <Counter increaseCount={increaseCount}
        decreaseCount={decreaseCount}
        countValue={countValue}/>
</Connect>
```

Компонент `Counter` получает доступ к функциям `increaseCount`, `decreaseCount` и `countValue`. Странно, что нет никакой функции рендеринга или эквивалентной ей. Все это автоматически обрабатывается React и его HOC-компонентом.

Мы почти закончили! Пришло время запустить компонент `Counter`. В каталоге *src* создайте файл *Counter.js*. Добавьте в него следующее содержимое:

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

class Counter extends Component {
  render() {
    return (
      <div className="container">
        <button className="buttons"
            onClick={this.props.decreaseCount}>-</button>
        <span>{this.props.countValue}</span>
        <button className="buttons"
            onClick={this.props.increaseCount}>+</button>
      </div>
    );
  }
}

export default Counter;
```

Вероятно, это будет самый скучный компонент, который вы видели за довольно долгое время. Мы уже говорили о том, как компонент `Connect` посылает свойства и другие связанные с ними махинации компоненту `Counter`. Вы можете увидеть эти свойства, используемые здесь, чтобы отобразить значение счетчика или вызвать соответствующую функцию при нажатии кнопки «плюс» или «минус».

Последнее, что нам нужно сделать, это создать CSS-файл для оформления счетчика. В папке *src*, с которой мы работали все это

время, создайте файл *index.css*. Откройте этот файл и добавьте в него следующие правила стиля:

```
body {
  margin: 0;
  padding: 0;
  font-family: sans-serif;
  display: flex;
  justify-content: center;
  background-color: #8E7C93;
}

.container {
  background-color: #FFF;
  margin: 100px;
  padding: 10px;
  border-radius: 3px;
  width: 200px;
  display: flex;
  align-items: center;
  justify-content: space-between;
}

.buttons {
  background-color: transparent;
  border: none;
  font-size: 16px;
  font-weight: bold;
  border-radius: 3px;
  transition: all .15s ease-in;
}

.buttons: hover: nth-child(1) {
  background-color: #F45B69;
}
```

```
.buttons: hover: nth-child(3) {
  background-color: #C0DFA1;
}
```

На этом мы закончили. Если вы еще этого не сделали, сохраните изменения во всех файлах, над которыми работали. Если вы запустите свое приложение в браузере (выполнив команду `npm start`), вы увидите, что счетчик работает так, как ожидалось.

Заключение

Во многих отношениях Redux предназначен для устранения некоторых недостатков, в которых React имеет преимущества. Мы рассмотрели некоторые из этих преимуществ, когда изучали, как должны передаваться данные в React. Даже можно сказать, что идеи, стоящие за Redux, должны быть уточнены как часть React. Но Redux тоже не идеален. Как и во многих вопросах программирования, Redux — это просто один из инструментов для выполнения задач. Не каждая ситуация с данными требует Redux. На самом деле Redux иногда может создать ненужные сложности в разработке проектов. Дэн Абрамов, один из создателей Redux, написал замечательную статью (**medium.com/@dan_abramov/you-mightnot-need-redux-be46360cf367**), описывающую некоторые ситуации, в которых, вероятно, не стоит использовать Redux для решения проблем. Я настоятельно рекомендую вам прочитать эту статью.

> Примечание: Если у вас возникнут какие-либо проблемы, задайте вопрос!
>
> Если у вас есть какие-либо вопросы или код не работает ожидаемым образом, не стесняйтесь задать вопрос! Опубликуйте свой вопрос на форуме **forum.kirupa.com** и получите помощь от самых дружелюбных и знающих людей в Интернете!

Предметный указатель

Научно-популярное издание

МИРОВОЙ КОМПЬЮТЕРНЫЙ БЕСТСЕЛЛЕР

Кирупа Чиннатамби

ИЗУЧАЕМ REACT

Главный редактор *Р. Фасхутдинов*
Руководитель направления *В. Обручев*
Ответственный редактор *Е. Минина*
Литературный редактор *М. Сюткина*
Младший редактор *Д. Атакишиева*
Художественный редактор *А. Шуклин*
Компьютерная верстка *Э. Брегис*
Корректор *Е. Сербина*

ООО «Издательство «Эксмо»
123308, Москва, ул. Зорге, д. 1. Тел.: 8 (495) 411-68-86.
Home page: www.eksmo.ru E-mail: info@eksmo.ru
Өндіруші: «ЭКСМО» АҚБ Баспасы, 123308, Мәскеу, Ресей, Зорге көшесі, 1 үй.
Тел.: 8 (495) 411-68-86.
Home page: www.eksmo.ru E-mail: info@eksmo.ru.
Тауар белгісі: «Эксмо»
Интернет-магазин : www.book24.ru

Интернет-магазин : www.book24.kz
Интернет-дукен : www.book24.kz
Импортёр в Республику Казахстан ТОО «РДЦ-Алматы».
Қазақстан Республикасындағы импорттаушы «РДЦ-Алматы» ЖШС.
Дистрибьютор и представитель по приему претензий на продукцию,
в Республике Казахстан: ТОО «РДЦ-Алматы»
Қазақстан Республикасында дистрибьютор және өнім бойынша арыз-талаптарды
қабылдаушының өкілі «РДЦ-Алматы» ЖШС,
Алматы қ., Домбровский көш., 3«а», литер Б, офис 1.
Тел.: 8 (727) 251-59-90/91/92; E-mail: RDC-Almaty@eksmo.kz
Өнімнің жарамдылық мерзімі шектелмеген.
Сертификация туралы ақпарат сайтта: www.eksmo.ru/certification

Сведения о подтверждении соответствия издания согласно законодательству РФ
о техническом регулировании можно получить на сайте Издательства «Эксмо»
www.eksmo.ru/certification
Өндірген мемлекет: Ресей. Сертификация қарастырылмаған

Подписано в печать 25.04.2019. Формат 70x100$^{1}/_{16}$.
Печать офсетная. Усл. печ. л. 29,81.
Тираж экз. Заказ

12+

EKSMO.RU
новинки издательства

ISBN 978-5-04-098028-4

КОГДА ВЫ ДАРИТЕ КНИГУ, ВЫ ДАРИТЕ ЦЕЛЫЙ МИР

ХОТИТЕ ЗНАТЬ БОЛЬШЕ?

Заходите на сайт:
https://eksmo.ru/b2b/

Звоните по телефону:
+7 495 411-68-59, доб. 2261

ВАШ ЛОГОТИП
НА ОБЛОЖКЕ

ВАШ ЛОГОТИП НА КОРЕШКЕ

ОБРАЩЕНИЕ
К КЛИЕНТАМ
НА ОБЛОЖКЕ

НАЧНИ СОЗДАВАТЬ СОВРЕМЕННЫЕ ВЕБ-ПРИЛОЖЕНИЯ С ПОМОЩЬЮ REACT

Даже на фоне феноменального развития современных фрейворков и библиотек для специалистов очевиден успех React. Он способен решить наиболее распространенные проблемы, с которыми сталкиваются разработчики при создании сложных приложений. Более того, React значительно упрощают создание визуальных элементов.

Считается, что React слишком сложен для начинающих. Когда-то это было правдой... До настоящего момента!

- Создайте свое первое приложение при помощи React
- Создавайте компоненты для определения пользовательского интерфейса
- Комбинируйте компоненты и совершенствуйте пользовательский интерфейс
- С JSX без труда определяйте визуальные элементы
- Поддерживайте работу приложения
- Узнайте о стилизации контента методом библиотеки React
- Познакомьтесь с методами жизненного цикла в действии
- Разработайте многостраничные приложения
- Используйте Redux, чтобы упростить работу с приложением

«Изучаем React» – первая обучающая книга по React, во время чтения которой вы начнете создавать собственные современные веб-приложения буквально с первых страниц. Это обновленное издание популярного у веб-разработчиков по всему миру самоучителя Кирупы Чиннатамби, автора Youtube-канала, посвященного веб-девелопменту для начинающих. Освойте самую популярную библиотеку JavaScript и шаблон Redux легко и быстро!

ISBN 978-5-04-098028-4