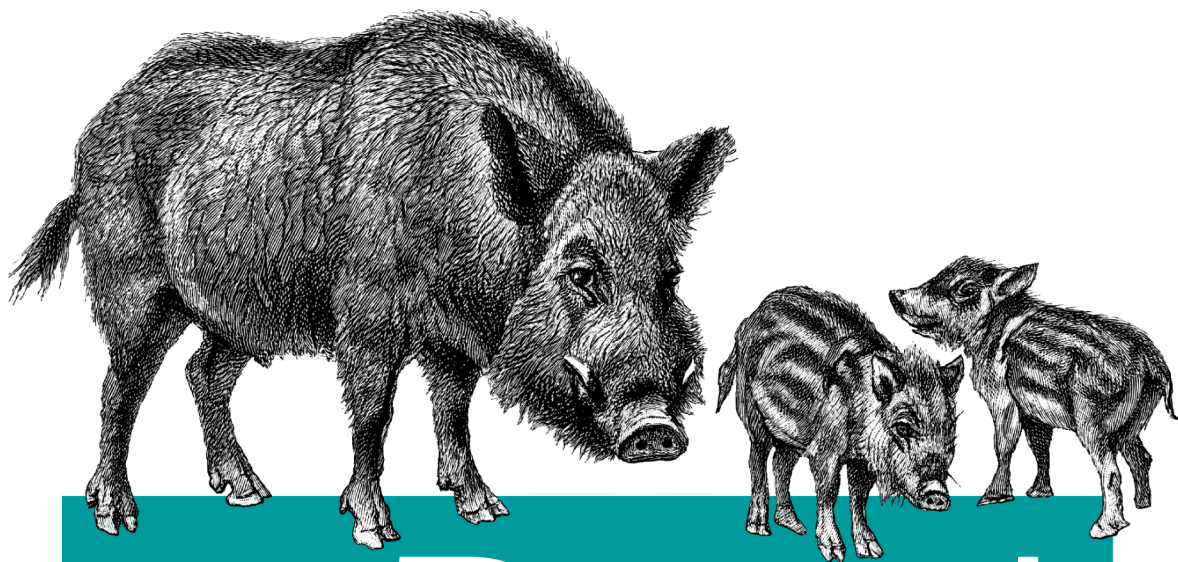


O'REILLY®



React и Redux

ФУНКЦИОНАЛЬНАЯ
ВЕБ-РАЗРАБОТКА

 ПИТЕР®

Алекс Бэнкс, Ева Порселло

Learning React

FUNCTIONAL WEB DEVELOPMENT
WITH REACT AND REDUX

Alex Banks and Eve Porcello

O'REILLY®

Beijing • Boston • Farnham • Sebastopol • Tokyo

React и Redux

ФУНКЦИОНАЛЬНАЯ
ВЕБ-РАЗРАБОТКА

Алекс Бэнкс, Ева Порселло



Санкт-Петербург · Москва · Екатеринбург · Воронеж
Нижний Новгород · Ростов-на-Дону
Самара · Минск

2018

БК 32.988.02-018
УДК 004.738.5
Б97

Бэнкс Алекс, Порселло Ева

Б97 React и Redux: функциональная веб-разработка. — СПб.: Питер, 2018. — 336 с.: ил. — (Серия «Бестселлеры O'Reilly»).

ISBN 978-5-4461-0668-4

Хотите научиться писать эффективные пользовательские интерфейсы при помощи React? Тогда вы нашли нужную книгу. Авторы расскажут, как создавать пользовательские интерфейсы при помощи этой компактной библиотеки и писать сайты, на которых можно обрабатывать огромные объемы данных без перезагрузки страниц. Также вы изучите новейшие возможности стандарта ECMAScript и функционального программирования.

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

БК 32.988.02-018
УДК 004.738.5

Права на издание получены по соглашению с O'Reilly. Все права защищены. Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме без письменного разрешения владельцев авторских прав.

Информация, содержащаяся в данной книге, получена из источников, рассматриваемых издательством как надежные. Тем не менее, имея в виду возможные человеческие или технические ошибки, издательство не может гарантировать абсолютную точность и полноту приводимых сведений и не несет ответственности за возможные ошибки, связанные с использованием книги.

ISBN 978-1491954621 англ.

Authorized Russian translation of the English edition of Learning React, ISBN 9781491954621 © 2017 Alex Banks, Eve Porcello
This translation is published and sold by permission of O'Reilly Media, Inc., which owns or controls all rights to publish and sell the same.

ISBN 978-5-4461-0668-4

© Перевод на русский язык ООО Издательство «Питер», 2018
© Издание на русском языке, оформление ООО Издательство «Питер», 2018
© Серия «Бестселлеры O'Reilly», 2018

Краткое содержание

Предисловие	11
Глава 1. Добро пожаловать в React	14
Глава 2. Новый синтаксис JavaScript	21
Глава 3. Функциональное программирование с применением JavaScript	42
Глава 4. Чистый React	70
Глава 5. React с JSX	91
Глава 6. Свойства, состояние и дерево компонентов	118
Глава 7. Усовершенствование компонентов	149
Глава 8. Redux.....	190
Глава 9. React Redux.....	217
Глава 10. Тестирование.....	234
Глава 11. Маршрутизатор React Router	282
Глава 12. React и сервер	304

Оглавление

Предисловие	11
Условные обозначения.....	11
Использование примеров кода.....	12
Благодарности.....	13
Глава 1. Добро пожаловать в React	14
Препятствия и трудности	15
React как библиотека	15
Новый синтаксис ECMAScript.....	15
Популярность функционального JavaScript	15
Утомительный JavaScript.....	16
Почему React будет не трудно изучить	16
Будущее библиотеки React.....	17
Нужно идти в ногу со временем	17
Работа с файлами	18
Файловый репозиторий	18
Инструменты React-разработчика	19
Установка Node.js.....	19
Глава 2. Новый синтаксис JavaScript	21
Объявление переменных в ES6	22
const	22
let.....	22
Шаблонные строки	24
Параметры по умолчанию	26
Стрелочные функции	26

Транспиляция ES6.....	30
Объекты и массивы ES6	31
Деструктурирующее присваивание.....	31
Расширения объектных литералов.....	33
Оператор распространения	34
Промисы	36
Классы	37
Модули ES6.....	39
CommonJS.....	41
Глава 3. Функциональное программирование с применением JavaScript.....	42
Значение понятия функциональности	43
Сравнение императивности с декларативностью.....	45
Функциональные концепции	47
Неизменяемость	47
Чистые функции.....	50
Преобразование данных.....	52
Функции высшего порядка	59
Рекурсия	60
Композиция.....	63
А теперь все вместе	65
Глава 4. Чистый React	70
Настройка страницы	70
Виртуальная DOM	71
Элементы React.....	73
ReactDOM.....	75
Дочерние элементы	76
Конструирование элементов с данными	78
Компоненты React.....	80
React.createClass.....	81
React.Component	83
Функциональные компоненты, не имеющие состояния	84
Отображение DOM	85
Фабрики.....	88

Глава 5. React с JSX	91
Элементы React в виде кода JSX	91
Советы по применению JSX.....	92
Вложенные компоненты	92
className.....	93
Выражения JavaScript	93
Вычисление.....	93
Отображение массивов на JSX.....	93
Babel.....	94
Введение в Webpack	103
Загрузчики Webpack	104
Приложение кулинарных рецептов с применением сборки, выполняемой Webpack	104
Глава 6. Свойства, состояние и дерево компонентов	118
Проверка свойств.....	118
Проверка свойств при использовании createClass	119
Свойства, используемые по умолчанию	123
Настраиваемая проверка свойств	124
Классы ES6 и функциональные компоненты, не имеющие состояния.....	125
Ссылки.....	128
Обратный поток данных	130
Ссылки в функциональных компонентах, не имеющих состояния.....	132
Управление состоянием React	133
Внедрение состояния компонента	133
Инициализация состояния из свойств.....	137
Состояние внутри дерева компонента	139
Новый взгляд на приложение организера цветов	140
Передача свойств вниз по дереву компонентов.....	141
Передача данных вверх по дереву компонентов.....	144
Глава 7. Усовершенствование компонентов	149
Жизненные циклы компонентов	149
Жизненный цикл установки.....	150

Жизненный цикл обновления	154
React.Children	164
Подключение библиотек JavaScript	166
Создание запросов с помощью Fetch	166
Подключение D3 Timeline	168
Компоненты высшего порядка	174
Управление состоянием за пределами React	180
Flux	182
Представление	184
Действия и создатели действий	185
Диспетчер	186
Хранилища	187
А теперь все вместе	188
Реализации Flux	188
Глава 8. Redux	190
Состояние	191
Действия	194
Преобразователи	197
Преобразователь цвета	200
Преобразователь цветов	202
Преобразователь сортировки	204
Хранилище	205
Подписка на хранилища	208
Сохранение в localStorage	209
Создатели действий	210
Функции промежуточного звена	213
Глава 9. React Redux	217
Явная передача хранилища	219
Передача хранилища через контекст	222
Сравнение презентационных и контейнерных компонентов	226
Провайдер React Redux	230
Функция connect библиотеки React Redux	231

Глава 10. Тестирование	234
ESLint.....	234
Тестирование Redux.....	238
Разработка, основанная на тестировании	239
Тестирование преобразователей	240
Тестирование хранилища	248
Тестирование компонентов React.....	251
Настройка среды Jest	251
Enzyme	252
Имитация компонентов.....	255
Тестирование на основе отображения мгновенного состояния (Snapshot Testing).....	263
Использование данных об охвате кода.....	268
Глава 11. Маршрутизатор React Router	282
Встраивание маршрутизатора	283
Вложенные маршруты.....	288
Использование страничного шаблона.....	288
Подразделы и подменю	290
Параметры маршрутизатора.....	294
Добавление страницы с информацией о цвете	295
Перемещение состояния сортировки цветов в маршрутизатор	300
Глава 12. React и сервер	304
Сравнение изоморфизма с универсализмом	305
Код React, отображаемый на сервере.....	308
Универсальный органайзер цветов	314
Универсальный Redux	316
Универсальная маршрутизация	318
Обмен данными с сервером.....	325
Выполнение действий на сервере.....	326
Действия с Redux Thunks.....	329

Предисловие

Эта книга предназначена для разработчиков, желающих изучить библиотеку React и освоить передовые технологии, внедряемые в язык JavaScript. Сейчас самое время стать JavaScript-разработчиком. Экосистема этого языка стремительно пополняется новыми инструментами, синтаксисом и наработанными методиками, обещающими разрешение множества проблем, стоящих перед разработчиками. Целью написания книги стала систематизация новых технологий, позволяющая сразу же приступить к работе с React. Мы рассмотрим библиотеку Redux, маршрутизатор React Router, а также инструменты разработчика и обещаем читателям, что не бросим их на произвол судьбы после того, как дадим вводную информацию.

Чтение книги не предполагает никаких предварительных знаний React. Все основы библиотеки будут представлены с самого начала. Мы также не строим никаких предположений относительно вашего опыта работы с ES6 или любым другим современным синтаксисом языка JavaScript. Мы дадим все это в главе 2 в качестве основы для понимания материала последующих глав.

Конечно, материал книги будет проще усвоить, имея достаточно ясное представление о том, что такое HTML, CSS и JavaScript. Прежде чем углубляться в изучение библиотек, лучше сперва как следует разобраться с этой большой тройцей.

В ходе изучения материала можно обращаться к хранилищу GitHub (<https://github.com/moonhighway/learning-react>). Там находится весь код, позволяющий освоить практические примеры.

Условные обозначения

В книге используются следующие типографские обозначения.

Курсив

Курсивом выделены новые термины.

Моноширинный шрифт

Используется для примеров программ, а также внутри абзацев, чтобы обратиться к элементам программы вроде переменных, функций, баз данных, типов

данных, переменных среды, инструкций и ключевых слов, имен файлов и расширений этих имен.

Моноширинный полужирный шрифт

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

Моноширинный курсивный шрифт

Показывает текст, который должен быть заменен значениями, введенными пользователем, или значениями, определяемыми контекстом.

Шрифт без засечек

Используется для обозначения URL, адресов электронной почты.



Этот рисунок указывает на совет или предложение.



Такой рисунок указывает на общее замечание.



Этот рисунок указывает на предупреждение.

Использование примеров кода

Сопроводительный материал (примеры кода, упражнения и т. д.) доступен для скачивания на <https://github.com/moonhighway/learning-react>.

Эта книга призвана помочь вам решать практические задачи. В целом, если пример кода предлагается вместе с книгой, вы можете использовать его в программах и документации. Вам не нужно обращаться к нам за разрешением, кроме случаев воспроизведения весьма существенного объема кода. Например, написание программы, использующей несколько фрагментов кода из этой книги, не требует разрешения, в отличие от продажи или распространения компакт-диска с примерами из книг издательства O'Reilly. Ответы на вопросы с цитированием материала данной книги и приведением примеров кода разрешения не требуют, чего не скажешь о вставке существенного объема примеров кода из этой книги в документацию по вашему программному продукту. Если есть сомнения по поводу правомерности использования примеров кода в рамках вышеизложенного, то можете свободно обратиться к нам за консультацией по адресу permissions@oreilly.com.

Благодарности

Наше путешествие по React вряд ли бы состоялось, не будь, как всегда, удачного стечения обстоятельств. Когда в стенах компании Yahoo! мы преподавали полный курс по разработке программ на JavaScript, для создания учебных материалов использовали библиотеку YUI. А затем, в августе 2014 года, разработка на YUI прекратилась. Нам пришлось заменить все файлы нашего курса, но чем? Что теперь предполагалось использовать при разработке внешнего интерфейса? Ответом была React. В нее мы влюбились не сразу, понадобилась пара часов на то, чтобы она зацепила нас. Казалось, React потенциально может изменить что угодно. Мы включились в работу с ней на ранней стадии и считаем, что нам очень повезло.

Эта книга не появилась бы на свет без участия Элли Макдональд (Ally MacDonald), помогавшей нам на всех этапах и весьма терпеливо перенесшей вместе с нами несколько обновлений библиотеки. Мы благодарны Мелани Ярбро (Melanie Yarbrough), Коллину Топорек (Colleen Toporek) и Рейчел Хед (Rachel Head) за их удивительное внимание к деталям. Спасибо Саре Ронау (Sarah Ronau) за корректуру этой книги непосредственно перед тем, как она готова была предстать перед читательским взором, и Бонни Эйзенман (Bonnie Eisenman) — за ценный совет и очаровательный характер. Спасибо также Стояну Стефанову (Stoyan Stefanov), который благосклонно согласился предоставить технический обзор, несмотря на большую занятость созданием передовых программных средств в компании Facebook.

Книга также не состоялась бы без участия Шэрон Адамс (Sharon Adams) и Мэрилин Мессинео (Marilyn Messineo). Именно они сговорились купить Алексу первый компьютер, который назывался Tandy TRS 80 Color Computer. И материал не превратился бы в книгу без любви, поддержки и поощрения со стороны Джима (Jim) и Лорри Порселло (Lorri Porcello), а также Майка (Mike) и Шэрон Адамс (Sharon Adams).

Хочется выразить благодарность и заведению Coffee Connection в Тахо-Сити, Калифорния, предоставлявшему нам кофе, столь необходимый для завершения работы над этой книгой, и его владельцу Робину, разразившемуся в наш адрес прописной, на его взгляд, истиной: «Книга по программированию? Звучит очень скучно!»

1 Добро пожаловать в React

React — популярная библиотека, используемая для создания пользовательских интерфейсов. Она была создана в Facebook с целью решить ряд проблем, связанных с крупномасштабными сайтами, управляемыми данными. В момент выпуска библиотеки в 2013 году этот проект воспринимался с некоторой долей скептицизма, поскольку соглашения по React довольно необычны.

В попытке не отпугнуть новых пользователей основная команда разработчиков React написала статью под названием *Why React?* (Почему именно React?), в которой рекомендовалось Give It (React) Five Minutes (Уделить ей (React) всего пять минут). Им хотелось сначала воодушевить людей на работу с библиотекой, прежде чем те подумают, что подход разработчиков слишком безумен.

React действительно является небольшой библиотекой, поставляемой без того набора инструментов, который может понадобиться для создания ваших приложений. Но уделите ей пять минут.

Да, благодаря React прямо в вашем JavaScript создается код, похожий на HTML. И разумеется, все эти теги требуют предварительной обработки перед запуском в браузере. А для этого, скорее всего, понадобится встроенный инструмент, такой как Webpack.

Если вы, как и мы, читали эту статью, то, возможно, были впечатлены промисами, связанными с новой библиотекой JavaScript, способной решить все проблемы с DOM (объектной моделью документа); с библиотекой, которая всегда будет проста в применении и никогда не причинит неудобств.

А потом начали возникать вопросы: как можно будет преобразовать этот JSX? Как загрузить данные? Куда девать CSS? Что такое декларативное программирование? В каждом направлении возникало еще больше вопросов о том, как внедрить эту библиотеку в повседневную работу. С каждым разговором появлялась новая терминология, возникали новые технические приемы и все больше вопросов.

Препятствия и трудности

Уделив несколько минут изучению компонентов React, вы откроете для себя новые горизонты осмысления подходов к веб-разработке. Разумеется, не обойдется и без трудностей, которые придется преодолеть, прежде чем приступить к созданию готового к практическому применению кода на основе React.

React как библиотека

Во-первых, стоит заметить, что библиотека невелика по размеру и используется только для одной части всей работы. Она не содержит инструментария, ожидаемого от традиционного фреймворка JavaScript. Основные решения о том, какими средствами экосистемы воспользоваться, принимают разработчики. Кроме того, постоянно появляются новые наборы инструментов, а старые отходят на второй план. При таком количестве библиотек совершенно не удивляет ощущение, что за ними всеми невозможно угнаться.

Новый синтаксис ECMAScript

React вступила в эпоху зрелости в весьма важный, но хаотичный период в истории JavaScript. Раньше Европейская ассоциация производителей компьютеров (ЕСМА) довольно редко выпускала новые спецификации. Иногда на выпуск одной из них уходило до десяти лет. Это значило, что разработчикам не нужно было очень часто осваивать новый синтаксис.

А с 2015 года новые функциональные свойства языка и синтаксические дополнения станут выпускаться каждый год. Номерные выпуски системы (ECMAScript 3, ECMAScript 5) будут заменены ежегодными (ECMAScript 2016, ECMAScript 2017). По мере развития языка первопроходцы сообщества React стремятся использовать новый синтаксис. Зачастую это говорит о том, что документация предполагает знание синтаксических новинок ECMAScript. Если вы не знакомы с самой последней спецификацией, то просмотр кода React может вызвать неуверенность в своих силах.

Популярность функционального JavaScript

Помимо изменений, появляющихся на уровне языка, существует также множество подвижек для функционального программирования на JavaScript. Заведомо функциональным языком JavaScript не является, но в его коде могут использоваться функциональные методы. В React придается особое значение приоритету функционального программирования над объектно-ориентированным. Этот сдвиг в мышлении может привести к существенным преимуществам в таких областях,

как пригодность к тестированию и повышение производительности. Но при таком количестве материалов по React, предполагающих осмысление парадигмы, усвоить сразу весь объем информации будет нелегко.

Утомительный JavaScript

Разговоры об утомительности JavaScript (<https://medium.com/@ericclemmons/javascript-fatigue-48d4011b6fc4>) превратились уже в некое клише, но такое мнение сложилось только из-за сложности сборки программного продукта. В прошлом файлы JavaScript просто добавлялись к вашей странице. Теперь же они должны пройти процесс сборки, обычно в виде автоматизированного непрерывного процесса поставки программного продукта. Имеется формируемый синтаксис, который нужно транспилировать для работы во всех браузерах. Имеется JSX, подлежащий преобразованию в JavaScript. Имеется SCSS, возможно требующий предварительной обработки. Эти компоненты нуждаются в тестировании, которое они должны успешно пройти. Вам могла бы понравиться React, но теперь еще нужно освоить Webpack, справиться с разбиением кода, его сжатием, тестированием и т. д. и т. п.

Почему React будет не трудно изучить

Цель данной книги — избежать путаницы в процессе обучения; мы даем материал в четкой последовательности, создавая прочную обучающую основу. Начнем с обновления синтаксиса, чтобы познакомить вас с последними свойствами JavaScript, особенно с теми, которые часто используются с React. Затем представим функциональный JavaScript, так что вы сможете сразу же применить эти методы и понять парадигму, породившую React.

Затем мы рассмотрим основополагающие сведения о React, вы сможете создать свои первые компоненты и получить ответы на вопросы, зачем и как нужно транспилировать код. После этого заложим площадку под новое приложение, позволяющее пользователям сохранять и упорядочивать цвета. Оно будет создано с использованием библиотеки React, затем код усовершенствуется благодаря ее новейшим методам. Мы расскажем о библиотеке Redux как о контейнере для данных клиента и завершим разработку приложения, внедрив Jest-тестирование и маршрутизацию с помощью средства React Router. В заключительной главе представим универсальный и структурно однородный код и усовершенствуем органайзер цветов, переведя его на сервер.

Надеемся на то, что ваша работа существенно ускорится благодаря применению экосистемы React наряду с подходом, предусматривающим не только поверхностное изучение, но и приобретение инструментов и навыков, необходимых для создания настоящих React-приложений.

Будущее библиотеки React

React пока не утратила своей новизны. Она дошла до стадии стабильности основных функциональных свойств, но даже они еще могут измениться. В следующие версии библиотеки будет включено средство Fiber — новая реализация основного алгоритма React, целью которого является увеличение скорости выдачи изображения на экран. Выстраивать предположения о том, насколько оно повлияет на React-разработчиков, пока рано, но оно, несомненно, отразится на скорости, с которой приложения выводят на экран изображение и обновляют его.

Многие из этих изменений связаны с целевыми устройствами. В данной книге рассматриваются способы разработки одностраничных веб-приложений с помощью React, но не нужно предполагать, что браузеры являются единственным местом, где могут работать такие приложения. Средство React Native, выпущенное в 2015 году, позволяет пользоваться преимуществами React-приложений в программах, предназначенных для работы под iOS и Android. Хотя об этом еще рано говорить, но уже появился фреймворк React VR для создания интерактивных приложений с поддержкой виртуальной реальности как способ разработки 360-градусных интерфейсов с использованием React и JavaScript. Команда библиотеки React настроит вас на быструю разработку восприятий для широкого диапазона размеров и типов экрана.

Мы надеемся предоставить вам достаточно прочную базу, позволяющую приспособиться к изменениям экосистемы и к созданию приложений, способных работать на платформах за пределами браузера.

Нужно идти в ногу со временем

По мере внесения изменений в React и сопутствующие инструменты иногда что-то перестает работать. Фактически некоторые из будущих версий данных утилит могут привести к неработоспособности ряда примеров, приводимых в данной книге. Но вы все равно можете придерживаться этих примеров. Точная информация о версии будет предоставлена в файле `package.json`, чтобы вы могли установить упомянутые там пакеты с нужной версией.

Кроме этой книги, обо всех последних изменениях можно узнать из новых публикаций в официальном блоге React (<https://facebook.github.io/react/blog/>). При выпуске новых версий React основная команда разработчиков опубликует там подробную информацию с описанием изменений и нововведений.

Существует также множество популярных React-конференций, которые можно посетить, чтобы получить самую свежую информацию о библиотеке. Если не получается принять в них личное участие, то можно изучить материалы сразу же после

проведения конференций, поскольку зачастую они публикуются на YouTube. К их числу относятся:

- ❑ *React Conf* (<http://conf.reactjs.org/>) — конференция, спонсируемая Facebook, в Санта-Кларе, Калифорния;
- ❑ *React Rally* (<http://www.reactrally.com/>) — конференция сообщества в Солт-Лейк-Сити;
- ❑ *ReactiveConf* (<https://reactiveconf.com/>) — конференция сообщества в Братиславе, Словакия;
- ❑ *React Amsterdam* (<https://react.amsterdam/>) — конференция сообщества в Амстердаме.

Работа с файлами

В данном разделе будут рассмотрены работа с файлами для этой книги и порядок установки некоторых полезных инструментов React.

Файловый репозиторий

GitHub-репозиторий, связанный с настоящей книгой (<https://github.com/moonhighway/learning-react>), предоставляет все файлы с программным кодом, систематизированные по главам. Репозиторий представлен сочетанием файлов с кодом и JSBin-образцов. JSBin — это интерактивный редактор кода, похожий на CodePen и JSFiddle.

Одно из основных преимуществ использования JSBin заключается в том, что после щелчка на ссылке можно сразу же экспериментировать с файлом. При создании JSBin-файла или в момент его исправления редактор сгенерирует уникальный URL для вашего примера кода, как показано на рис. 1.1.



Рис. 1.1. URL, созданный JSBin

Буквы после `jsbin.com` представляют собой уникальный ключ URL. После следующего слеша указан номер версии. В последней части URL будет указано одно из двух слов: `edit` для режима редактирования или `quiet` для режима предварительного просмотра.

Инструменты React-разработчика

Существует несколько инструментов, которые могут быть установлены в качестве расширений или дополнительных модулей браузера и представлять для вас определенный интерес:

- ❑ *react-detector* (<https://chrome.google.com/webstore/detail/react-detector/jaaklebbonndhkanegppscanebkdljh?hl=en-US>) — расширение браузера Chrome, позволяющее узнать, какие сайты используют React, а какие нет;
- ❑ *show-me-the-react* — еще одно средство, доступное для Firefox и Chrome, обнаруживающее React в ходе исследования ресурсов Интернета;
- ❑ *React Developer Tools* (рис. 1.2) (<https://github.com/facebook/react-devtools>) — дополнительный модуль, который может расширить функциональность инструментов разработчика браузера. Он создает новую вкладку в инструментах разработчика, где можно будет просматривать элементы React.

Если вы предпочитаете браузер Chrome, то этот модуль можно установить как расширение (<https://chrome.google.com/webstore/detail/react-developer-tools/fmkadmapgofadopljbjfkapdkoienihi>); его можно также установить в качестве дополнения для Firefox (<https://addons.mozilla.org/en-US/firefox/addon/react-devtools/>).

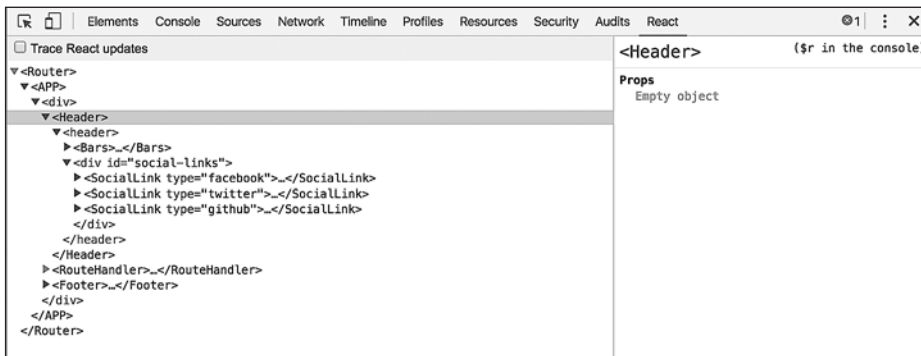


Рис. 1.2. Просмотр React Developer Tools

Всякий раз, когда вы увидите, что *react-detector* или *show-me-the-react* активны, можете открыть инструменты разработчика и получить представление о том, как React используется на сайте.

Установка Node.js

Node.js — JavaScript-язык без браузера. Это среда исполнения, применяемая для создания JavaScript-приложений на стороне как клиента, так и сервера. Node является средством с открытым кодом и может быть установлен на Windows, macOS, Linux и другие платформы. Мы прибегнем к Node в главе 12 при создании сервера Express.

Чтобы пользоваться React, Node не нужен. Но при работе с React для установки зависимостей необходимо будет задействовать диспетчер пакетов Node под названием `npm`. Он устанавливается автоматически вместе с Node.

Если вы не уверены в том, что на вашей машине установлен Node.js, то можно открыть терминал или окно командной строки и набрать команду:

```
$ node -v
Output: v7.3.0
```

В идеале у вас должен быть Node версии 4 или выше. Появившееся после набора команды сообщение об ошибке `Command not found` (Команда не найдена) означает, что среда не установлена. Установить ее можно непосредственно с сайта Node.js (<https://nodejs.org/en/>). Просто пройдите по автоматически предлагаемым этапам установки и, когда снова наберете команду `node -v`, увидите номер версии.

Управление зависимостями с помощью Yarn

Дополнительной альтернативой `npm` является Yarn. Этот инструмент был выпущен в 2016 году компанией Facebook в сотрудничестве с Exponent, Google и Tilde. Он помогает Facebook и другим компаниям надежнее управлять своими зависимостями; если использовать Yarn для установки пакетов, то можно заметить, что он работает намного быстрее. Производительность работы `npm` и Yarn можно сравнить на сайте Yarn (<https://yarnpkg.com/en/compare>).

Если вы знакомы с рабочим циклом `npm`, то воспользоваться ускорением, предлагаемым Yarn, будет очень просто. Сначала глобально установите Yarn с помощью `npm`.

```
npm install -g yarn
```

И после этого все будет готово к установке пакетов. При установке зависимостей из `package.json` вместо `npm install` можно запустить команду `yarn`.

При установке отдельных пакетов вместо запуска команды `npm install -save [имя_пакета]` запустите:

```
yarn add [имя_пакета]
```

Для удаления зависимости служит уже известная команда:

```
yarn remove [имя_пакета]
```

Yarn используется на практике компанией Facebook и включается в такие проекты, как React, ReactNative и create-react-app. Если попадется проект с файлом `yarn.lock`, значит, в нем задействован Yarn. По аналогии с командой `npm install`, все зависимости проекта можно установить, набрав команду `yarn install` или просто `yarn`.

Мы настроили среду React-разработки. Теперь можно начинать преодолевать учебные препятствия. В главе 2 мы рассмотрим ESMA и разберемся с самым последним синтаксисом JavaScript, наиболее часто встречающимся в коде React.

2

Новый синтаксис JavaScript

С момента появления в 1995 году JavaScript претерпел множество изменений. Сначала он существенно упростил добавление в веб-страницы интерактивных элементов. Затем, с появлением технологий DHTML и Ajax, стал более надежным. Теперь, с помощью Node.js, JavaScript превратился в язык, используемый для создания приложений на стороне как клиента, так и сервера. Комитетом, который занимался руководством ранее и руководит сейчас изменениями в JavaScript, является Европейская ассоциация производителей компьютеров (ЕСМА).

Изменения в языке инициируются сообществом. Все начинается с предложений, высказываемых его представителями. Присылать предложения комитету ЕСМА может кто угодно (<https://tc39.github.io/process-document/>). ЕСМА берет ответственность за управление и расстановку приоритетов по этим предложениям с целью решить, что включать в каждую спецификацию. Предложения принимаются, проходя четко определенные этапы, начиная с нулевого, на котором представлены самые свежие предложения, и до четвертого, представляющего готовые предложения.

Самое последнее крупное обновление спецификации было утверждено в июне 2015 года¹, и его называли по-всякому: ECMAScript 6, ES6, ES2015 и ES6Harmony. Текущими планами предусматривается ежегодный выпуск новой спецификации. Выпуск 2016 года был относительно небольшим по объему, но похоже, что в выпуск ES2017 будет включен ряд весьма полезных функций. Многие из них мы задействуем в книге и при любой возможности будем выбирать для использования новинки JavaScript.

Многие из этих новых возможностей уже поддерживаются самыми последними версиями браузеров. Кроме того, мы рассмотрим вопросы преобразования ново-

¹ Avram A. ECMAScript 2015 Has Been Approved. <https://www.infoq.com/news/2015/06/ecmascript-2015-es6>, InfoQ.

го синтаксиса JavaScript в синтаксис ES5, который уже сегодня будет работать практически во всех браузерах. Весьма полезным источником информации о функциональных новинках JavaScript и изменяющейся степени их поддержки браузерами может послужить таблица совместимости kangax (<http://kangax.github.io/compat-table/esnext/>).

В этой главе мы покажем все новинки JavaScript, которые станем повсеместно применять в книге. Если вы еще не перешли к использованию самых последних элементов синтаксиса, то сейчас самое время сделать это. Если же вы вполне освоились с особенностями языка ES.Next, то можете переходить к изучению следующей главы.

Объявление переменных в ES6

До появления ES6 единственным способом объявления переменной было использование ключевого слова `var`. Теперь же имеется ряд других вариантов, предоставляющих более широкие функциональные возможности.

`const`

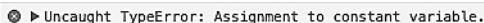
Константой считается переменная, значение которой не может быть изменено. Как это делалось и ранее в других языках программирования, с появлением ES6 в JavaScript были введены константы.

До появления констант мы располагали лишь переменными, а их значения могут перезаписываться:

```
var pizza = true
pizza = false
console.log(pizza) // false
```

Значение переменной, объявленной константой, переопределить нельзя, и при подобной попытке в консоль будет выдана ошибка (рис. 2.1):

```
const pizza = true
pizza = false
```



```
Uncaught TypeError: Assignment to constant variable.
```

Рис. 2.1. Попытка перезаписи константы

`let`

В JavaScript теперь имеется *лексическая область видимости переменных*. Блоки кода в этом языке создаются с помощью фигурных скобок (`{}`). Применяемые

с функциями, эти фигурные скобки изолируют область видимости переменных. В то же время нужно подумать об инструкциях `if-else`. Если до настоящего момента вы работали на других языках программирования, то можете предположить, что в этих блоках блокируется и область видимости переменных. Но это не так.

При создании переменной внутри блока `if-else` ее области видимости этим блоком не ограничиваются:

```
var topic = "JavaScript"

if (topic) {
  var topic = "React"
  console.log('block', topic) // block React
}

console.log('global', topic) // global React
```

Переменная `topic` внутри блока заново устанавливает значение `topic`.

Используя ключевое слово `let`, можно ограничить области видимости переменной любым блоком кода. Применение `let` защищает значение глобальной переменной:

```
var topic = "JavaScript"

if (topic) {
  let topic = "React"
  console.log('block', topic) // React
}

console.log('global', topic) // JavaScript
```

Значение `topic` за пределами блока заново не устанавливается.

Еще одной областью, где фигурные скобки не изолируют область видимости переменной, являются циклы `for`:

```
var div,
    container = document.getElementById('container')

for (var i=0; i<5; i++) {
  div = document.createElement('div')
  div.onclick = function() {
    alert('This is box #' + i)
  }
  container.appendChild(div)
}
```

В данном цикле создаются пять контейнеров `div` для появления внутри общего контейнера. Каждому из них определяется обработчик события щелчка `onclick`, открывающий окно сообщения, которое показывает индекс. Объявление `i` в цикле `for` приводит к созданию глобальной переменной `i`, а затем организуется итерация до тех пор, пока значение этой переменной не достигнет 5. При щелчке на каждом из этих прямоугольников в окне сообщения выводится предупреждение, что

`i` равно 5 для всех контейнеров `div`, поскольку текущим значением глобальной переменной `i` является 5 (рис. 2.2).

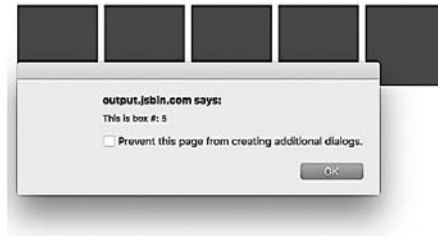


Рис. 2.2. Переменная `i` равна 5 для каждого прямоугольника

Объявление счетчика цикла `i` с применением ключевого слова `let` вместо `var` изолирует область видимости `i`. Теперь при щелчке на любом прямоугольнике будет отображаться значение `i`, ограниченное областью видимости итерации цикла (рис. 2.3):

```
var div, container = document.getElementById('container')
for (let i=0; i<5; i++) {
  div = document.createElement('div')
  div.onclick = function() {
    alert('This is box #: ' + i)
  }
  container.appendChild(div)
}
```

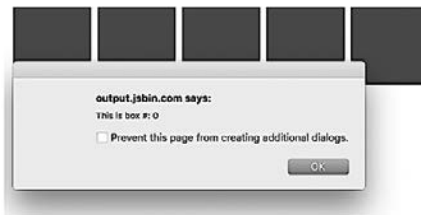


Рис. 2.3. Область видимости `i` защищена с помощью `let`

Шаблонные строки

Шаблонные строки представляют альтернативу объединению (конкатенации) строк. Кроме того, они позволяют вставлять в строки переменные.

При традиционном объединении строк для составления строки с использованием значений переменных и строк служат знаки «плюс» или запяты:

```
console.log(lastName + ", " + firstName + " " + middleName)
```

Применяя шаблон, можно создать одну строку и вставить значения переменных, заключив их в конструкцию `${ }`:

```
console.log(`${lastName}, ${firstName} ${middleName}`)
```

В строку шаблона между `${ }` и `}` может быть добавлен любой код JavaScript, возвращающий значение.

В строке шаблона остаются в неприкосновенности пробельные символы, упрощая тем самым подготовку шаблонов электронной почты, примеров кода или чего-нибудь еще, содержащего пробелы. Теперь можно располагать строкой, охватывающей несколько строк, не разбивая при этом создаваемый код. В примере 2.1 показано использование в шаблоне электронной почты символов табуляции, разрывов строк, пробелов и имен переменных.

Пример 2.1. Шаблонные строки оставляют пробельные символы в неприкосновенности

```

Hello ${firstName},

Thanks for ordering ${qty} tickets to ${event}.

Order Details
  ${firstName} ${middleName} ${lastName}
  ${qty} x ${price} = ${qty*price} to ${event}

You can pick your tickets up at will call 30 minutes before
the show.

Thanks,

${ticketAgent}

```

Ранее, при непосредственном использовании строки HTML в коде JavaScript, так легко сделать то же самое не удавалось, поскольку нужно было запускать все вместе в одной строке. Теперь, когда пробельные символы распознаются как текст, появляется возможность вставлять отформатированный HTML, который легко понять:

```
document.body.innerHTML = `
<section>
  <header>
    <h1>The HTML5 Blog</h1>
  </header>
  <article>
    <h2>${article.title}</h2>
    ${article.body}
  </article>
  <footer>
    <p>copyright ${new Date().getFullYear()} | The HTML5 Blog</p>
  </footer>
`
```

```
</section>
```

Обратите внимание: кроме всего прочего, можно включать переменные для заголовка страницы и текста статьи.

Параметры по умолчанию

Языки, включая C++ и Python, позволяют разработчикам объявлять для аргументов функций переменные, используемые по умолчанию. Параметры по умолчанию включены в спецификацию ES6, поэтому если значение для аргумента не предоставлено, то будет применено значение по умолчанию.

Например, можно настроить строки по умолчанию:

```
function logActivity(name="Shane McConkey", activity="skiing") {
  console.log( `${name} loves ${activity}` )
}
```

Если функции `favoriteActivity` не предоставить аргументы, то она отработает без сбоев, воспользовавшись значениями по умолчанию. Аргументы по умолчанию могут быть любого типа, а не просто строками:

```
var defaultPerson = {
  name: {
    first: "Shane",
    last: "McConkey"
  },
  favActivity: "skiing"
}

function logActivity(p=defaultPerson) {
  console.log( `${p.name.first} loves ${p.favActivity}` )
}
```

Стрелочные функции

Стрелочные функции — еще одна полезная новинка ES6. С их помощью можно создавать функции, не прибегая к ключевому слову `function`. Зачастую также не приходится использовать ключевое слово `return`. В примере 2.2 показан традиционный синтаксис функции.

Пример 2.2. Использование традиционной функции

```
var lordify = function(firstname) {
  return `${firstname} of Canterbury`
}

console.log( lordify("Dale") ) // Dale of Canterbury
console.log( lordify("Daryle") ) // Daryle of Canterbury
```

Используя стрелочную функцию, можно, как показано в примере 2.3, существенно упростить синтаксис.

Пример 2.3. Использование стрелочной функции

```
var lordify = firstname => `${firstname} of Canterbury`
```



Использование точки с запятой в этой книге

Точка с запятой является в JavaScript необязательным элементом. Наша философия такова: зачем ставить точку с запятой, когда она не требуется? В этой книге применяется принцип минимализма, исключающий использование ненужных элементов синтаксиса.

Теперь, используя стрелку, мы получаем всю функцию, уместившуюся в одной строке. Ключевое слово `function` удалено. Удалено и слово `return`, поскольку стрелка указывает на то, что должно быть возвращено. Еще одно преимущество заключается в том, что, если функция принимает только один аргумент, можно убрать и круглые скобки, окружающие аргументы.

Если применяется более одного аргумента, то их следует заключить в круглые скобки:

```
// Старый синтаксис
var lordify = function(firstName, land) {
  return `${firstName} of ${land}`
}

// Новый синтаксис
var lordify = (firstName, land) => `${firstName} of ${land}`
console.log( lordify("Dale", "Maryland") ) // Dale of Maryland
console.log( lordify("Daryle", "Culpeper") ) // Daryle of Culpeper
```

Функцию удалось выразить в одной строке, поскольку возвращению подлежал результат только одного выражения.

Если тело функции состоит более чем из одной строки, то его следует заключить в фигурные скобки:

```
// Старый синтаксис
var lordify = function(firstName, land) {

  if (!firstName) {
    throw new Error('A firstName is required to lordify')
  }

  if (!land) {
    throw new Error('A lord must have a land')
  }
}
```

```
    return `${firstName} of ${land}`
  }

  // Новый синтаксис
  var _lordify = (firstName, land) => {

    if (!firstName) {
      throw new Error('A firstName is required to lordify')
    }
    if (!land) {
      throw new Error('A lord must have a land')
    }

    return `${firstName} of ${land}`
  }

  console.log( lordify("Kelly", "Sonoma") ) // Kelly of Sonoma
  console.log( lordify("Dave") )           // ! ОШИБКА JAVASCRIPT
```

Имеющиеся инструкции `if-else` заключены в фигурные скобки, но все же еще выигрывают от более лаконичного синтаксиса стрелочной функции.

Стрелочные функции не изолируют ключевое слово `this`. Например, оно представляет в функции обратного вызова `setTimeout` нечто, но не объект `tahoe`:

```
var tahoe = {
  resorts: ["Kirkwood", "Squaw", "Alpine", "Heavenly", "Northstar"],
  print: function(delay=1000) {

    setTimeout(function() {
      console.log(this.resorts.join(", "))
    }, delay)

  }
}
```

```
tahoe.print() // Невозможно прочитать свойство 'join', принадлежащее undefined
```

Причиной выдачи данной ошибки является попытка использования метода `.join` для объекта, представленного ключевым словом `this`. В данном случае это объект `window`. Вместо приведенного кода можно воспользоваться синтаксисом стрелочной функции, защитив тем самым область видимости `this`:

```
var tahoe = {
  resorts: ["Kirkwood", "Squaw", "Alpine", "Heavenly", "Northstar"],
  print: function(delay=1000) {

    setTimeout(() => {
      console.log(this.resorts.join(", "))
    }, delay)

  }
}
```

```
tahoe.print() // Kirkwood, Squaw, Alpine, Heavenly, Northstar
```


Этот код выполняется правильно, и обращения можно объединить через запятые. Но об области видимости нужно помнить всегда. Стрелочные функции не изолируют область видимости `this`:

```
var tahoe = {
  resorts: ["Kirkwood", "Squaw", "Alpine", "Heavenly", "Northstar"],
  print: (delay=1000) => {

    setTimeout(() => {
      console.log(this.resorts.join(", "))
    }, delay)

  }
}

tahoe.print()
// Невозможно прочитать свойство resorts, принадлежащее undefined
```

Замена функции `print` на стрелочную функцию означает, что `this` фактически указывает на `window`.

Для проверки данного утверждения изменим сообщение, выводимое на консоль, с целью вычислить, что `this` — это `window`:

```
var tahoe = {
  resorts: ["Kirkwood", "Squaw", "Alpine", "Heavenly", "Northstar"],
  print: (delay=1000)=> {

    setTimeout(() => {
      console.log(this === window)
    }, delay)

  }
}

tahoe.print()
```

Выражение вычисляется в `true`. Чтобы зафиксировать `this`, можно воспользоваться обычной функцией:

```
var tahoe = {
  resorts: ["Kirkwood", "Squaw", "Alpine", "Heavenly", "Northstar"],
  print: function(delay=1000) {

    setTimeout(() => {
      console.log(this === window)
    }, delay)

  }
}

tahoe.print() // false
```

Транспиляция ES6

Синтаксис спецификации ES6 поддерживается не всеми браузерами, и даже те из них, где эта поддержка имеется, не поддерживают абсолютно все. Единственный способ гарантировать работоспособность вашего кода ES6 — преобразовать его перед запуском в браузере в код ES5. Этот процесс называется *транспиляцией*. Одним из наиболее популярных инструментов для транспиляции является Babel (<http://babeljs.io/>).

В прошлом воспользоваться новинками JavaScript можно было единственным способом: ждать неделями, месяцами или даже годами, пока браузеры не станут их поддерживать. Теперь же транспиляция позволяет применить новинки немедленно. Этап транспиляции делает JavaScript похожим на другие языки. Транспиляция не является компиляцией: наш код не компилируется в двоичный. Вместо этого он транспилируется в синтаксис, который может интерпретироваться более широким диапазоном браузеров. К тому же теперь у JavaScript имеется исходный код, а значит, будет ряд файлов, принадлежащих вашему проекту, не запускаемых в браузере. Фрагмент кода ES6 показан в примере 2.4. У нас имеется уже рассмотренная стрелочная функция в сочетании с отдельными аргументами по умолчанию для *x* и *y*.

Пример 2.4. Код ES6 до его транспиляции, выполняемой утилитой Babel

```
const add = (x=5, y=10) => console.log(x+y);
```

Запустив для этого кода транспилятор, на выходе мы получим следующий код:

```
"use strict";

var add = function add() {
  var x = arguments.length <= 0 || arguments[0] === undefined ?
    5 : arguments[0];
  var y = arguments.length <= 1 || arguments[1] === undefined ?
    10 : arguments[1];
  return console.log(x + y);
};
```

Транспилятор добавляет объявление `"use strict"` для запуска в строгом режиме. Значения по умолчанию для переменных *x* и *y* задаются с помощью массива `arguments`, технического приема, который может быть вам знаком. Получившийся код JavaScript пользуется более широкой поддержкой.

Транспиляцию JavaScript можно выполнить непосредственно в браузере, используя встраиваемый транспилятор Babel. Нужно просто включить файл `browser.js`, и любой сценарий, имеющий указание типа `type="text/babel"`, будет преобразован (хотя текущая версия Babel имеет порядковый номер 6, работать будет только CDN-сеть для Babel):

```
<script
  src="https://cdnjs.cloudflare.com/ajax/libs/babel-core/5.8.23/browser.js">
```

```
</script>
<script src="script.js" type="text/babel">
</script>
```



Транспиляция в браузере

Такой подход означает, что браузер совершает транспиляцию в ходе выполнения сценария. Этот замысел вряд ли подойдет для практического применения, поскольку работа приложения сильно замедлится. В главе 5 мы обсудим, как применять сценарии на практике. А пока CDN-ссылка позволит нам изучить функциональные свойства ES6 и воспользоваться ими.

Может прийти такая мысль: «Отлично! Когда ES6 станут поддерживать все браузеры, больше не придется задействовать Babel!» Но к тому времени, когда такое произойдет, мы захотим использовать возможности следующей версии спецификации. Пока не произойдет некий сдвиг тектонических плит, отказаться в обозримом будущем от применения Babel мы вряд ли сможем.

Объекты и массивы ES6

ES6 предоставляет новые способы работы с объектами и массивами и пути определения областей видимости переменных внутри этих наборов данных. К числу новых функций относятся деструктуризация, расширение объектных литералов и оператор распространения.

Деструктурирующее присваивание

Деструктурирующее присваивание позволяет задавать локальную видимость полей внутри объекта и объявлять те значения, которые будут использоваться.

Рассмотрим показанный ниже объект `sandwich`. У него четыре ключа, но нам нужно задействовать значения только двух из них. Для `bread` и `meat` можно задать область видимости для локального применения:

```
var sandwich = {
  bread: "dutch crunch",
  meat: "tuna",
  cheese: "swiss",
  toppings: ["lettuce", "tomato", "mustard"]
}
```

```
var {bread, meat} = sandwich
```

```
console.log(bread, meat) // dutch crunch tuna
```

Код вытаскивает `bread` и `meat` из объекта и создает для них локальные переменные. К тому же значения переменных `bread` и `meat` могут быть изменены:

```
var {bread, meat} = sandwich

bread = "garlic"
meat = "turkey"
console.log(bread) // garlic
console.log(meat) // turkey
console.log(sandwich.bread, sandwich.meat) // dutch crunch tuna
```

Деструктурировать можно также поступающие аргументы функций. Рассмотрим следующую функцию, которая будет регистрировать человека по имени, причисляя его к лордам:

```
var lordify = regularPerson => {
  console.log(`${regularPerson.firstname} of Canterbury`)
}

var regularPerson = {
  firstname: "Bill",
  lastname: "Wilson"
}

lordify(regularPerson) // Bill of Canterbury
```

Вместо того чтобы использовать синтаксис точечной нотации для проникновения в объект, можно деструктурировать те значения, которые нам нужно извлечь из элемента `regularPerson`:

```
var lordify = ({firstname}) => {
  console.log(`${firstname} of Canterbury`)
}

lordify(regularPerson) // Bill of Canterbury
```

Деструктуризация также придает коду наглядность, то есть он становится более описательным относительно того, что мы пытаемся выполнить. Путем деструктуризации фамилии (`firstname`) мы объявляем, что будем использовать только переменную `firstname`. Более подробно декларативное программирование рассмотрим в следующей главе.

Кроме того, значения могут быть деструктурированы из массивов. Предположим, что первое значение массива нужно присвоить имени переменной:

```
var [firstResort] = ["Kirkwood", "Squaw", "Alpine"]

console.log(firstResort) // Kirkwood
```

Используя *соответствие списку* с применением запятых, можно также пропустить ненужные значения. Соответствие происходит, когда запятые ставятся вместо тех элементов, которые должны быть пропущены. Применяя тот же самый массив, можно получить доступ к последнему значению, заменив первые два значения запятыми:

```
var [, , thirdResort] = ["Kirkwood", "Squaw", "Alpine"]

console.log(thirdResort) // Alpine
```

Чуть позже в этом разделе мы разовьем данный пример, скомбинировав деструктурирование массива и применение оператора распространения.

Расширения объектных литералов

Расширение объектных литералов является противоположностью деструктурирования. Это процесс реструктурирования или воссоединения. С его помощью можно превратить в объект переменные из глобальной области видимости:

```
var name = "Tallac"
var elevation = 9738

var funHike = {name,elevation}

console.log(funHike) // {name: "Tallac", elevation: 9738}
```

Теперь элементы `name` и `elevation` являются ключами объекта `funHike`.

Объектный литерал или реструктурирование поможет создать и методы объекта:

```
var name = "Tallac"
var elevation = 9738

var print = function() {
  console.log(`Mt. ${this.name} is ${this.elevation} feet tall`)
}

var funHike = {name,elevation,print}

funHike.print() // Mt. Tallac is 9738 feet tall
```

Обратите внимание: для доступа к ключам объекта использовалось ключевое слово `this`.

При определении методов объекта больше не нужно прибегать к ключевому слову `function` (пример 2.5).

Пример 2.5. Сравнение старого и нового синтаксиса: синтаксис объекта

```
// Старый синтаксис
var skier = {
  name: name,
  sound: sound,
  powderYell: function() {
    var yell = this.sound.toUpperCase()
    console.log(`${yell} ${yell} ${yell}!!!`)
  },
  speed: function(mph) {
    this.speed = mph
    console.log('speed:', mph)
  }
}
```

```
// Новый синтаксис
const skier = {
  name,
  sound,
  powderYell() {
    let yell = this.sound.toUpperCase()
    console.log(`${yell} ${yell} ${yell}!!!`)
  },
  speed(mph) {
    this.speed = mph
    console.log('speed:', mph)
  }
}
```

Расширение объектного литерала позволяет помещать глобальные переменные в объекты и сокращает объем набираемого кода за счет упразднения ключевого слова `function`.

Оператор распространения

Оператор распространения имеет вид многоточия (`...`) и выполняет несколько различных задач. Во-первых, позволяет объединять содержимое массивов. Например, если есть два массива, то можно создать третий, объединяющий их в один:

```
var peaks = ["Tallac", "Ralston", "Rose"]
var canyons = ["Ward", "Blackwood"]
var tahoe = [...peaks, ...canyons]
```

```
console.log(tahoe.join(', ')) // Tallac, Ralston, Rose, Ward, Blackwood
```

Все элементы из `peaks` и `canyons` помещены в новый массив под названием `tahoe`.

Посмотрим, как оператор распространения способен помочь справиться с задачей. Используя массив `peaks` из предыдущего примера, представим, что нам нужно получить последний, а не первый элемент массива. Поменять порядок следования элементов на обратный в сочетании с деструктурированием массива можно с помощью метода `Array.reverse`:

```
var peaks = ["Tallac", "Ralston", "Rose"]
var [last] = peaks.reverse()

console.log(last) // Rose
console.log(peaks.join(', ')) // Rose, Ralston, Tallac
```

Вы поняли, что произошло? Функция `reverse` фактически изменила массив, подвергла его мутации. В мире, где используется оператор, не нужно заставлять мутировать исходный массив; можно создать копию, а затем применить к ней реверсирование:

```
var peaks = ["Tallac", "Ralston", "Rose"]
var [last] = [...peaks].reverse()
```

```
console.log(last) // Rose
console.log(peaks.join(', ')) // Tallac, Ralston, Rose
```

Поскольку оператор распространения применялся для копирования массива, массив `peaks` остался в неприкосновенности и в последующем может применяться в своем первоначальном виде.

Оператор распространения может использоваться и для получения некоторых выбранных или остальных элементов массива:

```
var lakes = ["Donner", "Marlette", "Fallen Leaf", "Cascade"]

var [first, ...rest] = lakes

console.log(rest.join(", ")) // "Marlette, Fallen Leaf, Cascade"
```

Данный оператор может служить для сбора аргументов функции в массив. Создадим функцию, которая получает n аргументов с помощью оператора распространения, а затем использует их для вывода нескольких сообщений на консоль:

```
function directions(...args) {
  var [start, ...remaining] = args
  var [finish, ...stops] = remaining.reverse()

  console.log(`drive through ${args.length} towns`)
  console.log(`start in ${start}`)
  console.log(`the destination is ${finish}`)
  console.log(`stopping ${stops.length} times in between`)
}

directions(
  "Truckee",
  "Tahoe City",
  "Sunnyside",
  "Homewood",
  "Tahoma"
)
```

Функция `directions` получает аргументы, используя оператор распространения. Первый аргумент присваивается переменной `start`, последний — переменной `finish` с помощью метода `Array.reverse`. Затем применяется длина массива `arguments`, чтобы показать, через сколько городов мы проезжали. Количество остановок представлено длиной массива `arguments` за вычетом остановки `finish`. Тем самым обеспечивается невероятная гибкость, поскольку функцию `directions` можно задействовать для обработки любого количества остановок.

Оператор распространения может применяться и для объектов¹. Действие аналогично его использованию с массивами. В следующем примере будет показан тот же

¹ Свойства Rest/Spread. <https://github.com/tc39/proposals>.

способ, что и при объединении двух массивов в третий, но вместо массивов будут применяться объекты:

```
var morning = {
  breakfast: "oatmeal",
  lunch: "peanut butter and jelly"
}

var dinner = "mac and cheese"

var backpackingMeals = {
  ..morning,
  dinner
}

console.log(backpackingMeals) // {breakfast: "oatmeal",
                                lunch: "peanut butter and jelly",
                                dinner: "mac and cheese"}
```

Промисы

Промисы (иногда говорят «обещания») представляют способ разобраться в асинхронном поведении. При выдаче асинхронного запроса может произойти одно из двух: все пойдет так, как мы и надеялись, или же случится ошибка. Существует несколько различных типов успешных и неуспешных запросов. Например, можно попробовать несколько способов получения данных для достижения успеха. Кроме того, есть возможность получить несколько типов ошибок. Промисы предоставляют способ упростить возврат к простому прохождению или сбою.

Создадим асинхронный промис для загрузки данных из API `randomuser.me`. У него есть информация, похожая на адрес электронной почты, имя, номер телефона, место проживания и т. д. для мнимых сотрудников, и он является весьма подходящим источником фиктивных данных.

Функция `getFakeMembers` возвращает новый промис. Промис делает запрос к API. Если он завершается удачно, то данные будут загружены. В противном случае выдается ошибка:

```
const getFakeMembers = count => new Promise((resolves, rejects) => {
  const api = `https://api.randomuser.me/?nat=US&results=${count}`
  const request = new XMLHttpRequest()
  request.open('GET', api)
  request.onload = () =>
    (request.status === 200) ?
      resolves(JSON.parse(request.response).results) :
      reject(Error(request.statusText))
  request.onerror = (err) => rejects(err)
  request.send()
})
```


Данный код создает промис, но тот еще не используется. Применить его можно, вызвав функцию `getFakeMembers` и передав то количество сотрудников, которое должно быть загружено. Функция `then` может быть встроена в цепочку для выполнения каких-либо действий после выполнения промиса. Это называется композицией. Кроме того, для обработки ошибок служит дополнительная функция обратного вызова:

```
getFakeMembers(5).then(
  members => console.log(members),
  err => console.error(
    new Error("cannot load members from randomuser.me"))
)
```

Промисы облегчают работу с асинхронными запросами, и это хорошо, поскольку в JavaScript приходится работать с большим объемом асинхронно запрашиваемых данных. Активное применение промисов вам также встретится в Node.js, поэтому современному специалисту по JavaScript без твердых представлений о промисах не обойтись.

Классы

Ранее официально классы в JavaScript представлены не были. Типы определялись функциями. Приходилось создавать функцию, а затем определять методы для функции, используя прототип:

```
function Vacation(destination, length) {
  this.destination = destination
  this.length = length
}

Vacation.prototype.print = function() {
  console.log(this.destination + " | " + this.length + " days")
}

var maui = new Vacation("Maui", 7);

maui.print(); // Maui | 7
```

Тех, кто привык к классической объектной ориентации, такое положение дел, наверное, сводило с ума.

В ES6 вводится объявление класса, но JavaScript по-прежнему работает тем же способом. Функции являются объектами, а наследование обрабатывается прототипом, но если ранее вы занимались классической объектной ориентацией, то новый синтаксис приобретает более определенный смысл:

```
class Vacation {

  constructor(destination, length) {
    this.destination = destination
```

```
    this.length = length
  }

  print() {
    console.log(`${this.destination} will take ${this.length} days.`)
  }
}
```



Соглашение о применении прописных букв

Главное правило гласит: названия всех типов должны начинаться с прописных букв. Имена всех классов мы станем записывать, исходя из этого правила.

После создания класса можно ввести новый экземпляр класса, воспользовавшись ключевым словом `new`. Затем для класса можно вызвать пользовательский метод:

```
const trip = new Vacation("Santiago, Chile", 7);

console.log(trip.print()); // Chile will take 7 days.
```

После создания объекта класса можно использовать его всякий раз при необходимости создания новых экземпляров элемента `vacation`. Классы могут также расширяться. При этом потомок наследует свойства и методы родительского класса. Здесь свойствами и методами можно манипулировать, но по умолчанию все они будут унаследованы.

Классом `Vacation` можно воспользоваться в качестве абстрактного для создания различных типов отпусков. Например, чтобы добавить экипировку (`gear`), класс `Vacation` можно расширить классом `Expedition`:

```
class Expedition extends Vacation {

  constructor(destination, length, gear) {
    super(destination, length)
    this.gear = gear
  }

  print() {
    super.print()
    console.log(`Bring your ${this.gear.join(" and your ")}`)
  }
}
```

Это простое наследование: потомок наследует свойства родительского класса. Вызывая метод `printDetails`, принадлежащий классу `Vacation`, можно добавить новое содержимое к тому, что выводится методом `printDetails`, принадлежащим классу `Expedition`. Новый экземпляр создается абсолютно так же — через введение переменной и использование ключевого слова `new`:

```
const trip = new Expedition("Mt. Whitney", 3,
    ["sunglasses", "prayer flags", "camera"])

trip.print()
// Mt. Whitney will take 3 days.
// Bring your sunglasses and your prayer flags and your camera
```



Классы и прототипное наследование

Обращение к классу по-прежнему означает, что вы используете прототипное наследование JavaScript. При выводе на консоль `Vacation.prototype` можно заметить конструктор и методы `printDetails`, применяемые к прототипу.

Мы будем использовать классы лишь отчасти; основное внимание уделим функциональной парадигме. У классов есть и другие особенности, например, геттеры и сеттеры, а также статические методы, но в данной книге предпочтение отдается не объектно-ориентированным, а функциональным технологиям. Причина, по которой здесь представлен данный материал, заключается в том, что мы всем этим воспользуемся чуть позже при создании компонентов React.

Модули ES6

Модуль в JavaScript представляет собой фрагмент кода, пригодный для многократного использования, который может быть легко включен в другие файлы JavaScript. До последнего времени единственным способом работы с модульным JavaScript было включение библиотеки, способной импортировать и экспортировать модули. С появлением ES6 JavaScript поддерживает модули самостоятельно¹.

Модули JavaScript хранятся в отдельных файлах, по одному на модуль. При создании и экспорте модулей можно воспользоваться двумя вариантами: из одного модуля можно экспортировать сразу несколько объектов JavaScript или же экспортировать по одному объекту из каждого модуля.

В примере 2.6, `text-helpers.js`, экспортируются модуль и две функции.

Пример 2.6. Файл `./text-helpers.js`

```
export const print(message) => log(message, new Date())

export const log(message, timestamp) =>
    console.log(`${timestamp.toString()}: ${message}`)
```

¹ Mozilla Developer Network, JavaScript Code Modules. https://developer.mozilla.org/en-US/docs/Mozilla/JavaScript_code_modules.

Ключевое слово `export` может использоваться для экспортирования любого типа JavaScript, который будет применяться в другом модуле. В данном примере экспортируются функции `print` и `log`. Любые другие переменные в `text-helpers.js` будут иметь локальный характер по отношению к модулю.

Иногда может потребоваться экспортировать из модуля только одну переменную. В таких случаях можно воспользоваться инструкцией `export default` (пример 2.7).

Пример 2.7. Файл `./mt-freel.js`

```
const freel = new Expedition("Mt. Freel", 2, ["water", "snack"])

export default freel
```

Эта инструкция может служить вместо оператора `export`, когда требуется экспортировать только один тип. С другой стороны, и `export`, и `export default` могут применяться для любого типа JavaScript: основные элементы, объекты, массивы и функции¹.

Модули могут использоваться в других файлах JavaScript с помощью инструкции `import`. В модулях с несколькими экспортируемыми элементами можно обратиться к деструктурированию объектов. Модули, в которых применяется `export default`, импортируются в одну переменную:

```
import { print, log } from './text-helpers'
import freel from './mt-freel'

print('printing a message')
log('logging a message')

freel.print()
```

Для переменных модуля можно задать локальную область видимости под другими именами переменных:

```
import { print as p, log as l } from './text-helpers'

p('printing a message')
l('logging a message')
```

Кроме того, можно импортировать все в одну переменную с помощью знака `*`:

```
import * as fns from './text-helpers'
```

Модули ES6 поддерживаются еще не всеми браузерами. Однако в Babel такая поддержка есть, поэтому мы будем использовать их по всей книге.

¹ Mozilla Developer Network, Using JavaScript Code Modules. https://developer.mozilla.org/en-US/docs/Mozilla/JavaScript_code_modules/Using.

CommonJS

CommonJS — модульный шаблон, поддерживаемый всеми версиями Node.js¹. Можно по-прежнему использовать модули с Babel и Webpack. А благодаря CommonJS, как показано в примере 2.8, объекты JavaScript экспортируются с применением `module.exports`.

Пример 2.8. Файл `./txt-helpers.js`

```
const print(message) => log(message, new Date())

const log(message, timestamp) =>
  console.log(`${timestamp.toString()}: ${message}`)

module.exports = {print, log}
```

CommonJS не поддерживает инструкцию `import`. Вместо этого модули импортируются с помощью функции `require`:

```
const { log, print } = require('./txt-helpers')
```

Несомненно, JavaScript очень быстро развивается и адаптируется к растущим запросам, предъявляемым к языку. Функции ES6 и последующих спецификаций довольно быстро реализуются в браузерах, поэтому вполне разумно без всяких колебаний воспользоваться этими возможностями уже сейчас². Многие функции включены в спецификацию ES6 потому, что поддерживают технологии функционального программирования. В функциональном JavaScript код можно воспринимать как коллекцию функций, из которых можно составлять приложения. В следующей главе мы рассмотрим функциональные методы более подробно и обсудим причины, способные побудить вас к их использованию.

¹ Node.js Documentation, Modules. <https://nodejs.org/docs/latest/api/modules.html>.

² Для получения актуальной информации о совместимости обратитесь к таблице совместимости с ES6. <http://kangax.github.io/compat-table/es6/>.

3 Функциональное программирование с применением JavaScript

Приступая к исследованию области React-программирования, вы непременно заметите, что тема функционального программирования довольно часто выходит на первый план. Функциональные технологии используются все больше и больше в проектах JavaScript.

Возможно, вы, даже не особо задумываясь над этим, уже создавали функциональный код на JavaScript. Если вам приходилось отображать элементы массива или сокращать их количество, то, значит, вы уже были близки к тому, чтобы стать программистом, использующим функциональные технологии. Под функциональную парадигму JavaScript подпадают React, Flux и Redux. Умение разбираться в основных понятиях функционального программирования будет способствовать повышению ваших знаний по структурированию React-приложений.

Интерес к функциональному программированию появился в 1930-х годах с изобретением *лямбда-исчисления*, или λ -исчисления¹. Появившиеся в XVII веке функции были частью исчисления. Они могли отправляться другим функциям в качестве аргументов или возвращаться из функций в качестве результатов. Более сложные функции, называемые *функциями высшего порядка*, могли манипулировать функциями и использовать их в качестве либо аргументов, либо результатов, либо и того и другого. В 1930-х годах Алонзо Черч (Alonzo Church), находясь в Принстоне, проводил эксперименты с этими функциями высшего порядка, в ходе которых изобрел лямбда-исчисления.

¹ Scott D. S. λ -Calculus: Then & Now. http://turing100.acm.org/lambda_calculus_timeline.pdf.

В конце 1950-х Джон Маккарти (John McCarthy) воспользовался понятиями, получаемыми из λ -исчислений, и применил их к новому языку программирования Lisp. В этом языке была реализована концепция функций высшего порядка и функций, применяемых в качестве элементов или объектов первого класса (first-class members or first-class citizens). Функция считается *элементом первого класса*, когда может быть объявлена в качестве переменной и отправлена другим функциям в качестве аргумента. Такие функции могут быть даже возвращены из других функций.

В этой главе мы рассмотрим некоторые ключевые понятия функционального программирования и порядок реализации функциональных технологий на языке JavaScript.

Значение понятия функциональности

JavaScript поддерживает функциональное программирование, так как его функции относятся к объектам первого класса. Это значит следующее: функции могут делать то же самое, что и переменные. В спецификацию ES6 добавляются усовершенствования языка, позволяющие подкреплять методы функционального программирования. К ним относятся стрелочные функции, промисы и оператор распространения (см. главу 2).

В JavaScript функции могут представлять в ваших приложениях данные. Можно было заметить, что функции допустимо объявлять с использованием ключевого слова `var`, как это делается при объявлении строковых, числовых и других переменных:

```
var log = function(message) {
  console.log(message)
};

log("In JavaScript functions are variables")
// Функции в JavaScript являются переменными
```

Применяя синтаксис ES6, эту же функцию можно написать в виде стрелочной функции. Программисты, использующие функциональные технологии, создают множество небольших функций, и синтаксис стрелочных функций существенно облегчает работу:

```
const log = message => console.log(message)
```

Поскольку функции являются переменными, их можно добавлять к объектам:

```
const obj = {
  message: "They can be added to objects like variables",
  log(message) {
    console.log(message)
  }
}

obj.log(obj.message)
// Они, как и переменные, могут добавляться к объектам
```

Обе эти инструкции делают одно и то же: они сохраняют функцию в переменной по имени `log`. Кроме того, для объявления второй функции было использовано ключевое слово `const`, что предохранит ее от перезаписи.

Функции в JavaScript можно также добавлять к массивам:

```
const messages = [
  "They can be inserted into arrays",
  message => console.log(message),
  "like variables",
  message => console.log(message)
]

messages[1](messages[0]) // Они могут вставляться в массивы
messages[3](messages[2]) // как переменные
```

Функции, подобно всем другим переменным, могут отправляться другим функциям в качестве аргументов:

```
const insideFn = logger =>
  logger("They can be sent to other functions as arguments");

insideFn(message => console.log(message))
// Они могут отправляться другим функциям в качестве аргументов
```

Они точно так же, как и переменные, могут возвращаться из других функций:

```
var createScream = function(logger) {
  return function(message) {
    logger(message.toUpperCase() + "!!!")
  }
}

const scream = createScream(message => console.log(message))

scream('functions can be returned from other functions')
scream('createScream returns a function')
scream('scream invokes that returned function')
// ФУНКЦИИ МОГУТ ВОЗВРАЩАТЬСЯ ИЗ ДРУГИХ ФУНКЦИЙ!!!
// CREATESCREAM ВОЗВРАЩАЕТ ФУНКЦИЮ!!!
// SCREAM ВЫЗЫВАЕТ ВОЗВРАЩЕННУЮ ФУНКЦИЮ!!!
```

Последние два примера были функциями высшего порядка, то есть функциями, которые могут либо получать, либо возвращать другие функции. Используя синтаксис ES6, можно дать описание точно такой же функции высшего порядка `createScream` с помощью стрелок:

```
const createScream = logger => message =>
  logger(message.toUpperCase() + "!!!")
```

Отныне и впредь нужно будет обращать внимание на количество стрелок, используемых при объявлении функции. Наличие более одной стрелки означает присутствие функции высшего порядка.

Можно сказать, что JavaScript — функциональный язык, поскольку его функции относятся к объектам первого класса. То есть функции являются данными. Они могут сохраняться, извлекаться или проходить через ваши приложения точно так же, как и переменные.

Сравнение императивности с декларативностью

Функциональное программирование является частью более обширной парадигмы: *декларативного программирования*. Оно представляет собой отдельный стиль программирования. При его соблюдении приложения структурированы таким образом: описание того, что должно случиться, приоритетнее определения того, как это должно случиться.

Чтобы разобраться в сути декларативного программирования, сравним его с *императивным программированием*, или со стилем программирования, при котором вся забота сводится к тому, как достичь результатов с помощью кода. Рассмотрим типовую задачу: создание строкового значения, подходящего для использования в URL. Обычно такую задачу можно выполнить, заменив все пробелы в строке дефисами, поскольку применение пробелов в URL не допускается. Сначала рассмотрим императивный подход к решению этой задачи:

```
var string = "This is the midday show with Cheryl Waters";
var urlFriendly = "";

for (var i=0; i<string.length; i++) {
  if (string[i] === " ") {
    urlFriendly += "-";
  } else {
    urlFriendly += string[i];
  }
}

console.log(urlFriendly);
```

В этом примере выполняется циклический обход всех символов строки с заменой встречающихся пробелов. В структуре данной программы внимание уделяется исключительно тому, как эта задача может быть выполнена. Здесь используются цикл `for` и инструкция `if`, а также присваивание значений с помощью оператора равенства. Простое изучение кода не дает исчерпывающей информации. Чтобы можно было понять, чем они занимаются, императивные программы требуют большого объема комментариев.

А теперь посмотрим на декларативный подход к решению той же самой задачи:

```
const string = "This is the mid day show with Cheryl Waters"
const urlFriendly = string.replace(/ /g, "-")

console.log(urlFriendly)
```

Здесь вместе с регулярным выражением используется метод `string.replace`, заменяющий пробелы дефисами. Применение `string.replace` является способом описания предполагаемого исхода: пробелы в строке должны быть заменены. Подробности того, как обрабатываются пробелы, абстрагируются внутри функции `replace`. В декларативных программах то, что должно произойти, описывается самим синтаксисом, а подробности (как это должно произойти) абстрагируются.

В декларативных программах легко разобраться, ведь происходящее описывается самим кодом. Например, прочитайте синтаксис в следующем фрагменте кода — в нем подробно описано, что произойдет после загрузки участников (`members`) из API:

```
const loadAndMapMembers = compose(
  combineWith(sessionStorage, "members"),
  save(sessionStorage, "members"),
  scopeMembers(window),
  logMemberInfoToConsole,
  logFieldsToConsole("name.first"),
  countMembersBy("location.state"),
  prepStatesForMapping,
  save(sessionStorage, "map"),
  renderUSMap
);

getFakeMembers(100).then(loadAndMapMembers);
```

Декларативный подход более информативен, а следовательно, в нем проще разобраться. Подробности реализации каждой из этих функций абстрагированы. Эти небольшие функции имеют понятные имена и объединены таким образом, что получается описание, как именно данные об участниках проходят путь от загрузки к сохранению и к отображению на карте страны. Данный подход не требует подробного комментирования. Фактически декларативное программирование позволяет создавать приложения, в которых проще разобраться, а такое приложение проще масштабировать¹.

Рассмотрим задачу создания объектной модели документа, или DOM (<https://www.w3.org/DOM/>). Императивный подход будет нацелен на то, как именно построена DOM:

```
var target = document.getElementById('target');
var wrapper = document.createElement('div');
var headline = document.createElement('h1');

wrapper.id = "welcome";
headline.innerText = "Hello World";

wrapper.appendChild(headline);
target.appendChild(wrapper);
```

¹ Дополнительные сведения о парадигме декларативного программирования можно найти в статье «Википедии» Declarative Programming по адресу <http://wiki.c2.com/?DeclarativeProgramming>.

Этот код касается создания элементов, присваивания им значений и добавления их к документу. Когда DOM создана императивным способом, будет очень трудно вносить изменения, добавлять свойства или масштабировать 10 000 строк кода.

А теперь посмотрим, как можно выстроить DOM декларативным способом, используя компонент React:

```
const { render } = ReactDOM

const Welcome = () => (
  <div id="welcome">
    <h1>Hello World</h1>
  </div>
)

render(
  <Welcome />,
  document.getElementById('target')
)
```

React имеет декларативный характер. Здесь компонент `Welcome` дает описание DOM, которая должна быть выведена на экран. Для построения модели функция `render` использует инструкции, объявленные в компоненте, абстрагируя тем самым подробности того, как должна быть выведена DOM. Нам ясно дается понять, что нужно вывести наш компонент `Welcome` в элемент с идентификатором `ID`, имеющим значение `'target'`.

Функциональные концепции

После введения в функциональное программирование и разъяснения того, что означает «функциональное» или «декларативное», перейдем к введению в основные концепции функционального программирования: неизменяемость, чистоту, преобразование данных, функции высшего порядка и рекурсию.

Неизменяемость

Изменяемость предполагает возможность внесения изменений, следовательно, *неизменяемость* такую возможность не предоставляет. В функциональной программе данные являются неизменяемыми. Они никогда не изменяются.

Если вам нужно поделиться своим свидетельством о рождении с общественностью, но хочется отредактировать или убрать закрытую информацию, то у вас фактически есть два варианта: можно взять широкий маркер и закрасить им в оригинале свидетельства все закрытые данные, или же можно найти копировальный аппарат.

Предпочтительнее будет найти такой аппарат, снять копию свидетельства и закрасить маркером все закрытые данные на этой копии. Тогда у вас будет отредактированное свидетельство о рождении, готовое к выставлению на всеобщее обозрение, и оставшийся нетронутым его оригинал.

Именно так неизменяемые данные ведут себя в приложении. Вместо изменения исходных структур данных создаются измененные копии этих структур, используемые взамен оригинала.

Для понимания сути работы неизменяемости посмотрим, что означает изменение данных. Рассмотрим объект, представляющий цвет `lawn` (зеленая лужайка):

```
let color_lawn = {
  title: "lawn",
  color: "#00FF00",
  rating: 0
}
```

Можно создать функцию, оценивающую цвета, и использовать ее для изменения рейтинга объекта `color`:

```
function rateColor(color, rating) {
  color.rating = rating
  return color
}

console.log(rateColor(color_lawn, 5).rating) // 5
console.log(color_lawn.rating)              // 5
```

В JavaScript аргументы функции ссылаются на фактические данные. В такой установке рейтинга цвета нет ничего хорошего, поскольку она изменяет исходный объект. (Представьте, что вы поручили отредактировать и выставить на всеобщее обозрение свое свидетельство о рождении, а вам вернули оригинал с черными маркерными полосами, нанесенными поверх конфиденциальных сведений. Вы-то надеялись, что у исполнителей хватит здравого смысла скопировать ваше свидетельство о рождении и вернуть оригинал нетронутым.) Функцию `rateColor` можно переписать, чтобы она не наносила вред оригиналу (объекту `color`):

```
var rateColor = function(color, rating) {
  return Object.assign({}, color, {rating:rating})
}

console.log(rateColor(color_lawn, 5).rating) // 5
console.log(color_lawn.rating)              // 4
```

Здесь для изменения рейтинга цвета использовался метод `Object.assign`. Это «копировальная машина». Метод получает пустой объект, копирует в него цвет и переписывает рейтинг, применяя копию. Теперь у нас есть объект с новой оценкой цвета, и для этого не пришлось изменять оригинал.

Точно такую же функцию можно создать с помощью стрелочной функции ES6 и оператора распространения объекта ES7. Функция `rateColor` использует оператор распространения для копирования цвета в новый объект с последующей перезаписью его рейтинга:

```
const rateColor = (color, rating) =>
  ({
    ...color,
    rating
  })
```

Эта новая JavaScript-версия функции `rateColor` точно такая же, как и предыдущая. Она считает `color` неизменяемым объектом, при этом применяет более лаконичный синтаксис и выглядит намного понятнее. Обратите внимание: возвращаемый объект заключен в круглые скобки. При использовании стрелочной функции это обязательный шаг, поскольку стрелка не может просто указывать на фигурные стрелки объекта.

Рассмотрим массив названий цветов:

```
let list = [
  { title: "Rad Red"},
  { title: "Lawn"},
  { title: "Party Pink"}
]
```

Можно создать функцию, которая будет добавлять цвета к массиву, используя `Array.push`:

```
var addColor = function(title, colors) {
  colors.push({ title: title })
  return colors;
}

console.log(addColor("Glam Green", list).length) // 4
console.log(list.length) // 4
```

Но `Array.push` не является неизменяемой функцией. Это функция `addColor` изменяет исходный массив путем добавления к нему еще одного элемента. Чтобы сохранить неизменяемость массива `colors`, нужно воспользоваться функцией `Array.concat`:

```
const addColor = (title, array) => array.concat({title})

console.log(addColor("Glam Green", list).length) // 4
console.log(list.length) // 3
```

Функция `Array.concat` объединяет массивы. В данном случае она получает новый объект с новым названием цвета и добавляет его к копии исходного массива.

Оператор распространения ES6 можно использовать не только для копирования объектов, но и для объединения массивов. Вот как выглядит новый JavaScript-эквивалент предыдущей функции `addColor`:

```
const addColor = (title, list) => [...list, {title}]
```

Эта функция копирует исходный список в новый массив, а затем добавляет новый объект, содержащий название цвета к этой копии. Исходный массив она не изменяет.

Чистые функции

Чистой функцией называют функцию, которая возвращает значение, вычисляемое на основе ее аргументов. Чистые функции получают как минимум один аргумент и всегда возвращают значение или другую функцию. Эти функции не имеют побочных эффектов, не устанавливают значений глобальных переменных и не изменяют ничего, что относится к состоянию приложения. Они рассматривают свои аргументы в качестве неизменяемых данных.

Для понимания того, что такое чистые функции, сначала посмотрим на функцию с побочным эффектом:

```
var frederick = {
  name: "Frederick Douglass",
  canRead: false,
  canWrite: false
}

function selfEducate() {
  frederick.canRead = true
  frederick.canWrite = true
  return frederick
}

selfEducate()
console.log( frederick )
// {name: "Frederick Douglass", canRead: true, canWrite: true}
```

Функцию `selfEducate` нельзя считать чистой. Она не получает никаких аргументов и не возвращает значение или функцию. Она также изменяет переменную `Frederick` за пределами своей внутренней области видимости. При вызове функции `selfEducate` изменяется что-то в окружающем ее «мире». То есть она вызывает побочный эффект:

```
const frederick = {
  name: "Frederick Douglass",
  canRead: false,
  canWrite: false
}

const selfEducate = (person) => {
  person.canRead = true
  person.canWrite = true
  return person
}

console.log( selfEducate(frederick) )
console.log( frederick )
// {name: "Frederick Douglass", canRead: true, canWrite: true}
// {name: "Frederick Douglass", canRead: true, canWrite: true}
```



Тестируемость чистых функций

Чистые функции обладают свойством естественной тестируемости. Они не изменяют ничего касающегося своей среды окружения или «мира» и поэтому не требуют сложной настройки тестов или обособления. Ко всему, с чем чистая функция должна работать, она получает доступ через аргументы. При тестировании чистой функции осуществляется манипуляция аргументами, позволяющая оценивать конечный результат. Более подробные сведения о тестировании можно найти в главе 10.

Функцию `selfEducate` также нельзя отнести к чистой, поскольку она вызывает побочный эффект. Вызов этой функции приводит к изменениям переданных ей объектов. Чистая функция получается только в том случае, если передаваемые ей аргументы можно считать неизменяемыми.

Дадим этой функции аргумент:

```
const frederick = {
  name: "Frederick Douglass",
  canRead: false,
  canWrite: false
}

const selfEducate = person =>
  ({
    ...person,
    canRead: true,
    canWrite: true
  })

console.log( selfEducate(frederick) )
console.log( frederick )
// {name: "Frederick Douglass", canRead: true, canWrite: true}
// {name: "Frederick Douglass", canRead: false, canWrite: false}
```

В конечном итоге эта версия `selfEducate` стала чистой функцией. Она вычисляет значение на основе переданного ей аргумента `person`. Кроме того, возвращает новый объект `person` без изменения переданного ей аргумента, следовательно, не имеет никаких побочных эффектов.

Теперь изучим функцию с побочным эффектом, вносящую изменения в DOM:

```
function Header(text) {
  let h1 = document.createElement('h1');
  h1.innerText = text;
  document.body.appendChild(h1);
}

Header("Header() caused side effects");
```

Функция `Header` создает заголовок, то есть элемент с характерным текстом, и добавляет его к DOM. Это функция с побочным эффектом. Она не возвращает функцию или значение и создает побочный эффект: вносит изменение в DOM.

В React пользовательский интерфейс (UI) выражается с помощью чистых функций. В следующем фрагменте кода `Header` является чистой функцией, которой можно воспользоваться для создания заголовка, то есть точно такого же одного элемента, что и в предыдущем примере. Но сама по себе эта функция не вызывает никаких побочных эффектов, поскольку не вносит изменений в DOM. Эта функция создаст элемент, состоящий из одного заголовка первого формата, а использование данного элемента для изменения DOM будет возложено на какую-то другую часть приложения:

```
const Header = (props) => <h1>{props.title}</h1>
```

Чистые функции являются еще одной ключевой концепцией функционального программирования. Они существенно облегчат вашу работу, поскольку не станут влиять на состояние приложения. При написании функций старайтесь следовать трем правилам.

1. Функция должна получать как минимум один аргумент.
2. Функция должна возвращать значение или другую функцию.
3. Функция не должна вносить какие-либо изменения в переданные ей аргументы.

Преобразование данных

Но если данные являются неизменяемыми, то как изменяется что-либо в приложении? Функциональное программирование построено на преобразовании данных из одной формы в другую. Преобразованные копии будут создаваться с помощью функций. Эти функции снизят степень императивности кода, сделав его более простым.

Чтобы разобраться в способах создания одного набора данных на основе другого, никакая особая среда не понадобится. Все необходимые инструменты для решения этой задачи уже встроены в сам язык JavaScript. Для профессионального овладения функциональным JavaScript нужно освоить две ключевые функции: `Array.map` и `Array.reduce`.

В этом подразделе мы рассмотрим вопросы преобразования данных из одного типа в другой с помощью этих и некоторых других ключевых функций.

Рассмотрим массив, состоящий из названий гимназий:

```
const schools = [  
  "Yorktown",  
  "Washington & Lee",  
  "Wakefield"  
]
```

Воспользовавшись функцией `Array.join`, можно получить список с запятыми в качестве разделителей из этих и некоторых других строк:


```
console.log( schools.join(", ") )
// "Yorktown, Washington & Lee, Wakefield"
```

Метод `join` является встроенным в JavaScript методом работы с массивами, которым можно воспользоваться для извлечения из массива строки с разделителями. Исходный массив при этом остается в неприкосновенности, метод `join` просто использует его по-другому. Подробности создания данной строки абстрагируются от программиста.

Если нужна функция, создающая новый массив из названий гимназий, начинающихся на букву `W`, то можно воспользоваться методом `Array.filter`:

```
const wSchools = schools.filter(school => school[0] === "W")
console.log( wSchools )
// ["Washington & Lee", "Wakefield"]
```

Функция `Array.filter` встроена в JavaScript и создает новый массив из массива-источника. В качестве своего единственного аргумента она получает *предикат*. Так называется функция, которая всегда возвращает булево значение: `true` или `false`. Функция `Array.filter` вызывает этот предикат по одному разу для каждого элемента массива. Данный элемент передается предикату в качестве аргумента, а возвращаемое значение используется для принятия решения по включению элемента в новый массив. В данном случае `Array.filter` проверяет название каждой гимназии, чтобы посмотреть, не начинается ли оно с буквы `W`.

При решении задачи по удалению элемента из массива нужно пользоваться функцией `Array.filter`, а не функцией `Array.pop` или `Array.splice`, так как `Array.filter` не вносит в данные никаких изменений. В следующем фрагменте кода функция `cutSchool` возвращает новый массив, не пропуская через фильтр указанные названия гимназий:

```
const cutSchool = (cut, list) =>
  list.filter(school => school !== cut)

console.log(cutSchool("Washington & Lee", schools).join(" * "))
// "Yorktown * Wakefield"

console.log(schools.join("\n"))
// Yorktown
// Washington & Lee
// Wakefield
```

В данном случае функция `cutSchool` используется для возвращения нового массива, который не содержит `Washington & Lee`. Затем для этого нового массива применяется функция `join`, чтобы создать строку со звездочкой в качестве разделителя из оставшихся двух названий гимназий. Функция `cutSchool` является чистой. Она получает список гимназий и то название гимназии, которое должно быть удалено, и возвращает новый массив без указанного названия гимназии.

Еще одной функцией для работы с массивами, имеющей большое значение для функционального программирования, является `Array.map`. Вместо предиката метод

`Array.map` получает в качестве своего аргумента функцию. Эта функция будет вызываться по одному разу для каждого элемента массива, и то, что она возвращает, будет добавлено к новому массиву:

```
const highSchools = schools.map(school => `${school} High School`)  
  
console.log(highSchools.join("\n"))  
// Yorktown High School  
// Washington & Lee High School  
// Wakefield High School  
  
console.log(schools.join("\n"))  
// Yorktown  
// Washington & Lee  
// Wakefield
```

В данном случае функция `map` использовалась для добавления к каждому названию гимназии строки `High School`. При этом массив `schools` остался в неприкосновенности.

В последнем примере массив строк был создан из массива строк. Функция `map` может создавать массив из объектов, значений, массивов, других функций, то есть из любого имеющегося в JavaScript типа. Рассмотрим пример функции `map`, возвращающей объект для каждой гимназии:

```
const highSchools = schools.map(school => ({ name: school })))  
  
console.log( highSchools )  
// [  
// { name: "Yorktown" },  
// { name: "Washington & Lee" },  
// { name: "Wakefield" }  
// ]
```

Массив содержит объекты, созданные из массива, который содержит строки.

Если нужно создать чистую функцию, изменяющую один объект в массив объектов, то для этого также может использоваться функция `map`. В следующем примере имя гимназии `Stratford` будет изменено на `HB Woodlawn` без внесения изменений в массив `schools`:

```
let schools = [  
  { name: "Yorktown"},  
  { name: "Stratford" },  
  { name: "Washington & Lee"},  
  { name: "Wakefield"}  
]  
  
let updatedSchools = editName("Stratford", "HB Woodlawn", schools)  
  
console.log( updatedSchools[1] ) // { name: "HB Woodlawn" }  
console.log( schools[1] )       // { name: "Stratford" },
```

Массив `schools` является массивом объектов. Переменная `updatedSchools` вызывает функцию `editName`, которой передается требующая обновления гимназия, новое название гимназии и массив `schools`. Тем самым изменения вносятся в новый массив без редактирования исходного массива:

```
const editName = (oldName, name, arr) =>
  arr.map(item => {
    if (item.name === oldName) {
      return {
        ...item,
        name
      }
    } else {
      return item
    }
  })
```

Внутри `editName` функция `map` используется для создания нового массива объектов на основе исходного массива. Индекс каждого элемента вставляется в функцию обратного вызова методом `Array.map` в качестве второго аргумента, переменной `i`. Когда значение `i` не равно индексу редактируемого элемента, точно такой же элемент просто помещается в новый массив. Когда же значение `i` равно этому индексу, элемент с таким индексом заменяется в новом массиве новым объектом.

Функция `editName` может быть написана целиком в одной строке. Вот как выглядит пример той же самой функции с использованием сокращенной формы инструкции `if-else`:

```
const editName = (oldName, name, arr) =>
  arr.map(item => (item.name === oldName) ?
    ({...item,name}) :
    item
  )
```

Если нужно преобразовать массив в объект, можно воспользоваться методом `Array.map` в сочетании с `Object.keys`. Последний может использоваться для возвращения из объекта массива ключей.

Предположим, что нужно преобразовать объект `schools` в массив `schools`:

```
const schools = {
  "Yorktown": 10,
  "Washington & Lee": 2,
  "Wakefield": 5
}

const schoolArray = Object.keys(schools).map(key =>
  ({
    name: key,
    wins: schools[key]
  })
)
```

```
console.log(schoolArray)
// [
//   {
//     name: "Yorktown",
//     wins: 10
//   },
//   {
//     name: "Washington & Lee",
//     wins: 2
//   },
//   {
//     name: "Wakefield",
//     wins: 5
//   }
// ]
```

В данном примере `Object.keys` возвращает массив названий гимназий, в отношении которого можно применить функцию `map`, чтобы создать новый массив той же длины. Имя нового объекта будет установлено с помощью ключа, а количество побед (`wins`) будет задано равным значению.

Итак, мы узнали, что преобразование массивов можно выполнить с применением `Array.map` и `Array.filter`. Мы также узнали, что превратить массивы в объекты можно благодаря использованию сочетания `Object.keys` и `Array.map`. И последним инструментом, необходимым для нашего функционального арсенала, станет возможность преобразовывать массивы в элементарные типы и другие объекты.

Для преобразования массива в любое значение, включая число, строку, булево значение, объект или даже функцию, могут использоваться функции `reduce` и `reduceRight`.

Предположим, что в массиве из чисел нужно найти самое большое число. Массив нужно преобразовать в число, для чего можно воспользоваться функцией `reduce`:

```
const ages = [21,18,42,40,64,63,34];

const maxAge = ages.reduce((max, age) => {
  console.log(`${age} > ${max} = ${age > max}`);
  if (age > max) {
    return age
  } else {
    return max
  }
}, 0)

console.log('maxAge', maxAge);
// 21 > 0 = true
// 18 > 21 = false
// 42 > 21 = true
// 40 > 42 = false
// 64 > 42 = true
// 63 > 64 = false
// 34 > 64 = false
// maxAge 64
```

Массив возрастов `ages` был сведен к одному значению — максимальному возрасту, 64. Функция `reduce` получает два аргумента: функцию обратного вызова и исходное значение. В данном случае исходным значением является `0`, в результате чего оно устанавливается для начального максимума. Функция обратного вызова вызывается по одному разу для каждого элемента массива. При первом вызове этой функции значение `age` равно 21, то есть значению первого элемента массива, а значение `max` равно `0`, то есть исходному значению. Функция обратного вызова возвращает большее из двух чисел, 21, которое становится значением `max` в ходе следующей итерации. При каждой итерации каждое значение `age` сравнивается со значением `max` и возвращается большее из двух значений. Наконец, последнее число, имеющееся в массиве, сравнивается с числом, возвращенным из предыдущего вызова функции обратного вызова.

Если из предыдущей функции убрать инструкцию `console.log` и воспользоваться сокращенной формой инструкции `if-else`, то максимальное значение в массиве можно будет вычислить с применением следующего синтаксиса:

```
const max = ages.reduce(
  (max, value) => (value > max) ? value : max,
  0
)
```



Array.reduceRight

Функция `Array.reduceRight` работает аналогично функции `Array.reduce`. Различие заключается в том, что она начинает сокращение не с начала, а с конца массива.

Иногда нам нужно превратить массив в объект. В следующем примере функция `reduce` используется для преобразования массива, содержащего названия цветов, в хеш:

```
const colors = [
  {
    id: '-xekare',
    title: "rad red",
    rating: 3
  },
  {
    id: '-jbwsf',
    title: "big blue",
    rating: 2
  },
  {
    id: '-prigbj',
    title: "grizzly grey",
    rating: 5
  },
  {
```

```

        id: '-ryhbhsl',
        title: "banana",
        rating: 1
    }
]

const hashColors = colors.reduce(
  (hash, {id, title, rating}) => {
    hash[id] = {title, rating}
    return hash
  },
  {}
)

console.log(hashColors);
// {
//   "-xekare": {
//     title:"rad red",
//     rating:3
//   },
//   "-jbwsof": {
//     title:"big blue",
//     rating:2
//   },
//   "-prigbj": {
//     title:"grizzly grey",
//     rating:5
//   },
//   "-ryhbhsl": {
//     title:"banana",
//     rating:1
//   }
// }

```

В данном примере второй аргумент, переданный функции `reduce`, является пустым объектом. Это наше исходное значение для хеша. В ходе каждой итерации функция обратного вызова добавляет к хешу новый ключ, используя форму записи с квадратными скобками, и задает значение для данного ключа по значению поля идентификатора массива `id`. Таким образом, функция `Array.reduce` может использоваться для сведения массива к одному значению, в данном случае к объекту.

Применяя `reduce`, массивы можно даже преобразовать в совершенно другие массивы. Рассмотрим сведение массива с несколькими экземплярами одного и того же значения в массив различных значений. Для выполнения этой задачи можно воспользоваться методом `reduce`:

```

const colors = ["red", "red", "green", "blue", "green"];

const distinctColors = colors.reduce(
  (distinct, color) =>
    (distinct.indexOf(color) !== -1) ?
      distinct :
      [...distinct, color],
  []
)

```

```
console.log(distinctColors)
// ["red", "green", "blue"]
```

В данном примере массив `colors` был сведен к массиву различных значений. Вторым аргументом, переданным функции `reduce`, являлся пустой массив. Он послужил исходным значением для `distinct`. Если в массиве `distinct` еще не содержится указанный цвет, то он будет в него добавлен. В противном случае он будет пропущен и возвратится массив `distinct` в его текущем состоянии.

Функции `map` и `reduce` — основное оружие любого функционального программиста, и JavaScript не исключение. Тем, кто желает стать профессиональным специалистом по JavaScript, эти функции следует освоить досконально. Возможность создания одного набора данных из другого требует определенного мастерства и может пригодиться для парадигмы программирования любого типа.

Функции высшего порядка

Использование *функций высшего порядка* также играет важную роль для функционального программирования. Эти функции уже упоминались и даже кое-где применялись в данной главе. Они способны манипулировать другими функциями и могут получать функции в качестве аргументов, или возвращать функции, или делать и то и другое.

К первой категории функций высшего порядка относятся функции, которые ожидают в качестве аргументов другие функции. Все следующие функции: `Array.map`, `Array.filter` и `Array.reduce` — получают функции в качестве аргументов. Поэтому они считаются функциями высшего порядка¹.

Посмотрим на способ реализации функции высшего порядка. В следующем примере будет создана функция обратного вызова `invokeIf`, проверяющая условие и вызывающая функцию обратного вызова, когда условие вычисляется в `true`, и другую функцию обратного вызова, когда это условие вычисляется в `false`:

```
const invokeIf = (condition, fnTrue, fnFalse) =>
  (condition) ? fnTrue() : fnFalse()

const showWelcome = () =>
  console.log("Welcome!!!")

const showUnauthorized = () =>
  console.log("Unauthorized!!!")
invokeIf(true, showWelcome, showUnauthorized) // "Welcome"
invokeIf(false, showWelcome, showUnauthorized) // "Unauthorized"
```

Функция `invokeIf` ожидает передачи ей двух функций: одной для `true` и другой для `false`. Это показано путем передачи функции `invokeIf` как функции `showWelcome`,

¹ Дополнительные сведения о функциях высшего порядка изложены в главе 5 книги *Eloquent JavaScript*. http://eloquentjavascript.net/05_higher_order.html.

так и функции `showUnauthorized`. Когда условие вычисляется в `true`, вызывается функция `showWelcome`. А когда в `false` — вызывается функция `showUnauthorized`.

Функции высшего порядка, возвращающие другие функции, могут быть полезны при решении трудностей, связанных с применением в JavaScript асинхронного режима работы. Они способны помочь в создании функций, которыми можно будет воспользоваться однократно или повторно по мере надобности.

Каррирование — функциональная технология, для которой привлекается использование функций высшего порядка. При каррировании практикуется удержание отдельных значений, необходимых для завершения операции, до тех пор пока чуть позже не сможет быть предоставлено все остальное. Это достигается путем применения функции, возвращающей другую функцию, называемую каррированной.

Рассмотрим показанный ниже пример каррирования. Функция `userLogs` зависит от некоей информации (от имени пользователя) и возвращает функцию, которая может использоваться однократно и повторно, при доступности всей остальной информации (сообщения). В данном примере все регистрационные сообщения будут предваряться связанным с ними именем пользователя. Обратите внимание на применение нами функции `getFakeMembers`, возвращающей промис, рассмотренный в главе 2:

```
const userLogs = userName => message =>
  console.log(` ${userName} -> ${message} `)

const log = userLogs("grandpa23")

log("attempted to load 20 fake members")
  getFakeMembers(20).then(
    members => log(` successfully loaded ${members.length} members `),
    error => log("encountered an error loading members")
  )
// grandpa23 -> попытка загрузки 20 мнимых сотрудников
// grandpa23 -> успешно загружены 20 сотрудников
// grandpa23 -> попытка загрузки 20 мнимых сотрудников
// grandpa23 -> при загрузке сотрудников возникла ошибка
```

Функция `userLogs` относится к функциям высшего порядка. Функция `log` создается из `userLogs`, и при каждом использовании функции `log` перед сообщением ставится строка `grandpa23`.

Рекурсия

Рекурсией называется технология создания функций, вызывающих самих себя. Зачастую, сталкиваясь с трудностями, связанными с циклами, можно вместо цикла воспользоваться рекурсивной функцией. Рассмотрим задачу обратного отсчета от числа 10. Для ее решения можно создать цикл `for` или же в качестве альтернативного варианта воспользоваться рекурсивной функцией. В следующем примере функция обратного отсчета `countdown` является рекурсивной:


```

const countdown = (value, fn) => {
  fn(value)
  return (value > 0) ? countdown(value-1, fn) : value
}

countdown(10, value => console.log(value));
// 10
// 9
// 8
// 7
// 6
// 5
// 4
// 3
// 2
// 1
// 0

```

В качестве аргументов функция `countdown` ожидает получения числа и функции. В данном примере она вызывается со значением `10` и функцией обратного вызова. При вызове `countdown` вызывается функция обратного вызова, которая выводит в консоль текущее значение. Затем `countdown` проверяет значение, чтобы определить, не больше ли оно нуля. Если больше, то `countdown` вызывает саму себя со значением, уменьшенным на единицу. Со временем значение станет равным нулю и `countdown` возвратит его обратно по всему пути стека вызовов.



Ограничения стека вызовов браузера

Рекурсия должна использоваться вместо циклов везде, где только возможно, но не все движки JavaScript оптимизированы под большое количество рекурсий. Слишком большое число рекурсий может привести к ошибкам JavaScript. Появления этих ошибок можно избежать, внедряя передовые методы очистки стека вызовов и «сплющивания» рекурсивных вызовов. В будущих движках JavaScript ограничения по стеку вызовов планируется полностью снять.

Рекурсия — еще одна функциональная технология, хорошо работающая с асинхронными процессами. Функции могут вызывать самих себя, если готовы к этому.

Функция `countdown` может быть изменена для выполнения обратного отсчета с задержкой. Эта модифицированная версия функции `countdown` может использоваться для создания таймера с обратным отсчетом:

```

const countdown = (value, fn, delay=1000) => {
  fn(value)
  return (value > 0) ?
    setTimeout(() => countdown(value-1, fn), delay) :
    value
}

```

```
const log = value => console.log(value)
countdown(10, log);
```

В данном примере путем начального однократного вызова `countdown` с числом 10 в функции, выводящей в консоль обратный отсчет, был создан десятисекундный таймер с обратным отсчетом. Вместо немедленного вызова самой себя функция `countdown` выжидает перед таким вызовом одну секунду, в результате чего получается таймер.

Рекурсия нашла хорошее применение в поиске структур данных. Ее можно применять для обхода подпапок, пока не будет идентифицирована папка, содержащая одни только файлы. Рекурсию можно также использовать для последовательного обхода DOM HTML до тех пор, пока не будет найден элемент, не содержащий дочерних элементов. В следующем примере рекурсия применена для глубокого обхода объекта с целью извлечения вложенного значения:

```
var dan = {
  type: "person",
  data: {
    gender: "male",
    info: {
      id: 22,
      fullname: {
        first: "Dan",
        last: "Deacon"
      }
    }
  }
}

deepPick("type", dan); // "person"
deepPick("data.info.fullname.first", dan); // "Dan"
```

Функция `deepPick` может служить для доступа к типу `Dan`, хранящемуся сразу же в первом объекте, или для углубления во вложенные объекты для нахождения имени `Dan`. Отправляя строку, в которой используется точечная нотация, можно указать, где найти значения, глубоко вложенные в объект:

```
const deepPick = (fields, object={}) => {
  const [first, ...remaining] = fields.split(".")
  return (remaining.length) ?
    deepPick(remaining.join("."), object[first]) :
    object[first]
}
```

Функция `deepPick` собирается либо вернуть значение, либо вызвать саму себя, пока в конце концов не возвратит значение. Сначала эта функция разбивает записанные с применением точек поля на массив и для отделения первого значения от оставшихся использует деструктуризацию массива. Если значения остались, то `deepPick` вызывает саму себя со слегка отличающимися данными, позволяя углубиться еще на один уровень.

Эта функция продолжает вызывать саму себя, пока в строке полей не останется точек; это будет означать, что полей больше не осталось. В следующем фрагменте кода можно увидеть, как по мере итераций, выполняемых функцией `deepPick`, изменяются значения для `first`, `remaining` и `object[first]`:

```
deepPick("data.info.fullname.first", dan); // "Deacon"
// Первая итерация
// first = "data"
// remaining.join(".") = "info.fullname.first"
// object[first] = { gender: "male", {info} }

// Вторая итерация
// first = "info"
// remaining.join(".") = "fullname.first"
// object[first] = {id: 22, {fullname}}

// Третья итерация
// first = "fullname"
// remaining.join(".") = "first"
// object[first] = {first: "Dan", last: "Deacon" }

// Наконец...
// first = "first"
// remaining.length = 0
// object[first] = "Deacon"
```

Рекурсия является весьма эффективной и легко реализуемой функциональной технологией. Используйте ее вместо организации циклов везде, где только возможно.

Композиция

В функциональных программах логика разбивается на небольшие чистые функции, нацеленные на решение конкретных задач. Со временем эти небольшие функции требуется собрать воедино. В частности, может понадобиться объединить их, вызвать последовательно или параллельно, скомпоновать в более крупные функции, пока в конечном итоге не получится приложение.

При составлении композиций используется множество различных реализаций, шаблонов и технологий. Одним из, возможно, уже известных вам приемов является выстраивание цепочки. Функции в JavaScript могут быть составлены в цепочку с помощью системы записи, применяющей точки, чтобы действие выполнялось над значением, возвращенным предыдущей функцией.

У строк имеется метод замены `replace`. Он возвращает шаблонную строку, у которой также имеется этот метод. Следовательно, для преобразования строки методы `replace` можно выстроить в цепочку с применением точечной нотации.

```
const template = "hh:mm:ss tt"
const clockTime = template.replace("hh", "03")
    .replace("mm", "33")
```

```
.replace("ss", "33")
.replace("tt", "PM")

console.log(clockTime)
// "03:33:33 PM"
```

В этом примере шаблоном является строка. Пристраивая цепочку методов замены к концу строки шаблона, можно заменить в строке часы, минуты, секунды и время суток новыми значениями. Сам шаблон остается нетронутым и может повторно применяться для создания дополнительных отображений показаний часов.

Составление в цепочку — один из методов создания композиции, но наряду с ним существуют и другие. Целью составления композиции является «создание функции более высокого порядка путем объединения более простых функций»¹.

```
const both = date => appendAMPM(civilianHours(date))
```

Функция `both` относится к функциям, пропускающим значение по конвейеру через две отдельные функции. Данные на выходе `civilianHours` становятся входом для `appendAMPM`, и отсчет времени можно изменить, используя обе эти функции, объединенные в одну. Но в этом синтаксисе трудно разобраться, а следовательно, могут возникнуть осложнения с его поддержкой и масштабированием. Представляете, что получится, когда понадобится переправить значение через 20 различных функций?

Более рациональный подход предусматривает создание функции высшего порядка, чем можно воспользоваться для составления композиций из функций и помещения их в более крупные функции.

```
const both = compose(
  civilianHours,
  appendAMPM
)
```

```
both(new Date())
```

Этот подход выглядит намного лучше. При нем не создаются трудности для масштабирования, поскольку добавлять дополнительные функции можно в любое место. Он также облегчает внесение изменений в порядок следования функций, из которых составляется композиция.

Функция-композиция относится к функциям высшего порядка. Она получает функции в качестве аргументов, а возвращает одно значение.

```
const compose = (...fns) =>
  (arg) =>
    fns.reduce(
      (composed, f) => f(composed),
      arg
    )
```

¹ Functional.js Composition. <http://functionaljs.com/functions/compose/>.

Функция `compose` получает в качестве аргументов функции и возвращает одну функцию. В этой реализации оператор распространения использован для превращения этих функций-аргументов в массив по имени `fns`. Затем возвращается функция, ожидающая один аргумент, `arg`. Когда вызывается эта функция, массив `fns` выстраивается в конвейер, начинающийся с аргумента, который нужно пропустить через функции. Аргумент становится исходным значением для `composed`, а затем при каждой итерации возвращается урезанная функция обратного вызова. Обратите внимание на то, что функция обратного вызова получает два аргумента: `composed` и функцию `f`. Каждая функция вызывается с аргументом `composed`, являющимся результатом, полученным на выходе предыдущей функции. В конечном итоге будет вызвана последняя функция и возвращен последний результат.

Этот простой пример функции-композиции разработан для демонстрации технологии составления композиции. Когда приходится обрабатывать более одного аргумента или иметь дело с аргументами, не являющимися функциями, эта функция усложняется. В других реализациях композиции¹ может использоваться функция `reduceRight`, которая будет составлять функции в обратном порядке.

А теперь все вместе

После введения в основные концепции функционального программирования найдем им практическое применение и создадим небольшое JavaScript-приложение.

Поскольку JavaScript позволяет отступать от функциональной парадигмы, не обязывая следовать правилам, вам нужно будет сосредоточиться. Соблюдение трех простых правил поможет не сбиваться с курса.

1. Сохраняйте неизменяемость данных.
2. Сохраняйте чистоту функций, пусть они получают как минимум один аргумент и возвращают данные или другую функцию.
3. Используйте рекурсию вместо циклов (везде, где только возможно).

Наша задача — создать тикающие часы. Они должны отображать часы, минуты, секунды и время суток в обычном гражданском формате. У каждого поля всегда должна быть пара цифр, то есть перед одиночной цифрой, 1 или 2, должен быть начальный ноль. Часы также должны тикать и изменять отображение каждую секунду.

Сначала рассмотрим императивное решение этой задачи.

```
// Вывод показаний часов каждую секунду
setInterval(logClockTime, 1000);
```

```
function logClockTime() {
```

¹ Еще одна реализация композиции представлена в Redux. <http://redux.js.org/docs/api/compose.html>.

```
// Получение строки показания часов в гражданском формате
var time = getClockTime();

// Очистка показаний консоли и вывод показания часов
console.clear();
console.log(time);
}

function getClockTime() {
  // Получение текущего времени
  var date = new Date();
  var time = "";

  // Выстраивание последовательности показания часов
  var time = {
    hours: date.getHours(),
    minutes: date.getMinutes(),
    seconds: date.getSeconds(),
    ampm: "AM"
  }

  // Преобразование показания времени в гражданский формат
  if (time.hours == 12) {
    time.ampm = "PM";
  } else if (time.hours > 12) {
    time.ampm = "PM";
    time.hours -= 12;
  }

  // Подстановка 0 к показанию часов, чтобы получалась пара цифр
  if (time.hours < 10) {
    time.hours = "0" + time.hours;
  }

  // Подстановка 0 к показанию минут, чтобы получалась пара цифр
  if (time.minutes < 10) {
    time.minutes = "0" + time.minutes;
  }

  // Подстановка 0 к показанию секунд, чтобы получалась пара цифр
  if (time.seconds < 10) {
    time.seconds = "0" + time.seconds;
  }

  // Придание показаниям часов формата строки "hh:mm:ss tt"
  return time.hours + ":"
    + time.minutes + ":"
    + time.seconds + " "
    + time.ampm;
}
```

Это весьма прямолинейное и работоспособное решение, а комментарии помогают разобраться в происходящем. И тем не менее все эти функции раздуты и сложны. Каждая выполняет слишком много действий. В них довольно трудно разобраться, они требуют комментариев, и их трудно сопровождать. А теперь посмотрим,

как с помощью функционального подхода можно получить более масштабируемое приложение.

Наша цель — разбить логику приложения на более мелкие части, то есть на функции. Каждая из них будет нацелена на решение всего одной задачи, и из таких функций составятся более крупные функции-композиции, которые можно использовать для создания часов.

Сначала создадим ряд функций, предоставляющих значения и управляющих консолью. Нам понадобится функция, выдающая секунду, функция, выдающая текущее время, и пара функций, выводящих сообщения на консоль и очищающих ее. В функциональных программах везде, где только возможно, вместо значений будут использоваться функции. При необходимости получить значение мы станем вызывать функцию.

```
const oneSecond = () => 1000
const getCurrentTime = () => new Date()
const clear = () => console.clear()
const log = message => console.log(message)
```

Затем нам понадобится ряд функций для преобразования данных. Эти три функции будут использоваться для изменения объекта `Date` в объект, который может быть применен в наших часах:

- ❑ `serializeClockTime` — получает объект времени и возвращает объект для показания часов, содержащих часы, минуты и секунды;
- ❑ `civilianHours` — получает объект показания часов и возвращает объект, где показание часов преобразовано к формату гражданского времени. Например: 1300 превращается в 1 час;
- ❑ `appendAMPM` — получает объект показания часов и добавляет к нему время суток, АМ или РМ.

```
const serializeClockTime = date =>
  ({
    hours: date.getHours(),
    minutes: date.getMinutes(),
    seconds: date.getSeconds()
  })

const civilianHours = clockTime =>
  ({
    ...clockTime,
    hours: (clockTime.hours > 12) ?
      clockTime.hours - 12 :
      clockTime.hours
  })

const appendAMPM = clockTime =>
  ({
    ...clockTime,
    ampm: (clockTime.hours >= 12) ? "PM" : "AM"
  })
```

Эти три функции используются для преобразования данных без изменения исходного значения. В них аргументы рассматриваются как неизменяемые объекты.

Затем нам понадобятся несколько функций высшего порядка.

- ❑ `display` — получает функцию цели `target` и возвращает функцию, которая будет отправлять время в адрес цели. В данном примере целью будет `console.log`.
- ❑ `formatClock` — получает шаблонную строку и использует ее для возвращения показания времени, отформатированного по критериям, заданным строкой. В данном примере шаблон имеет вид `hh:mm:ss tt`. Таким образом, `formatClock` будет заменять заполнители показаниями часов, минут, секунд и времени суток.
- ❑ `prependZero` — получает в качестве аргумента ключ объекта и ставит ноль впереди значения, хранящегося под этим ключом объекта. Функция получает ключ к указанному полю и ставит перед значениями ноль, если значение меньше 10.

```
const display = target => time => target(time)

const formatClock = format =>
  time =>
    format.replace("hh", time.hours)
      .replace("mm", time.minutes)
      .replace("ss", time.seconds)
      .replace("tt", time.ampm)

const prependZero = key => clockTime =>
  ({
    ...clockTime,
    [key]: (clockTime[key] < 10) ?
      "0" + clockTime[key] :
      clockTime[key]
  })
```

Эти функции высшего порядка будут вызываться для создания функций, многократно использующихся для форматирования показания времени для каждого тика. Обе функции — и `formatClock`, и `prependZero` — будут вызываться единожды для начальной настройки требуемого шаблона или ключа. Возвращаемые ими внутренние функции будут вызываться один раз в секунду для форматирования отображаемого времени.

Теперь, когда у нас есть все функции, требуемые для создания тикающих часов, нужно будет составить из них композицию. Для этого мы используем функцию-композицию, которая была определена в последнем разделе.

- ❑ `convertToCivilianTime` — отдельная функция, получающая в качестве аргумента показание времени и преобразующая его в формат гражданского времени с помощью обеих форм этого времени.
- ❑ `doubleDigits` — отдельная функция, получающая в качестве аргумента показание времени и обеспечивающая отображение часов, минут и секунд парой цифр, подставляя для этого ноль, где необходимо.

- `startTicking` — запускает часы, устанавливая интервал, вызывающий функцию обратного вызова каждую секунду. Функция обратного вызова представляет собой композицию из всех наших функций. Каждую секунду консоль очищается, получается текущее время, показание которого проходит преобразование, перевод в гражданский формат, форматирование и отображение.

```
const convertToCivilianTime = clockTime =>
  compose(
    appendAMPM,
    civilianHours
  )(clockTime)

const doubleDigits = civilianTime =>
  compose(
    prependZero("hours"),
    prependZero("minutes"),
    prependZero("seconds")
  )(civilianTime)

const startTicking = () =>
  setInterval(
    compose(
      clear,
      getCurrentTime,
      serializeClockTime,
      convertToCivilianTime,
      doubleDigits,
      formatClock("hh:mm:ss tt"),
      display(log)
    ),
    oneSecond()
  )
startTicking()
```

Эта декларативная версия часов достигает тех же результатов, что и императивная версия. Но у данного подхода имеется множество преимуществ. Во-первых, все эти функции намного проще тестируются и гораздо больше подходят для многократного использования. Они могут применяться в других вариантах часов или для другого цифрового отображения. Кроме того, эта программа легче масштабируется. У нее нет побочных эффектов. За пределами самих функций нет никаких глобальных переменных. Ошибки, конечно, могут допускаться, но их будет гораздо легче обнаружить.

В этой главе мы представили принципы функционального программирования. Когда в данной книге будут рассматриваться оптимальные методы, используемые в React и Flux, мы продолжим показывать, как эти библиотеки основаны на применении функциональных методов. В следующей главе мы приступим к углубленному изучению библиотеки React с более четким пониманием тех принципов, которыми руководствовались ее разработчики.

4 Чистый React

Для демонстрации того, как React работает в браузере, в данной главе вся работа будет вестись исключительно с этой библиотекой. Введение в JSX или в JavaScript как в XML мы оставим для следующей главы. Вполне вероятно, что ранее вам уже приходилось работать с библиотекой, даже не заглядывая в чистый код React, который создается в ходе транспиляции JSX в React. Библиотекой можно вполне успешно пользоваться, даже не заглядывая в чистый код React. Но если уделить время изучению всего того, что происходит «за кулисами», можно достичь более эффективных результатов, особенно когда дело дойдет до отладки кода. Нашей целью в этой главе будет заглянуть «за кулисы» и понять, как работает React.

Настройка страницы

Чтобы работать с React в браузере, нужно включить две библиотеки: React и ReactDOM. Первая представляет собой библиотеку для создания представлений, а вторая — библиотеку для фактического отображения пользовательского интерфейса в браузере.

ReactDOM

Для версии 0.14 React и ReactDOM были разбиты на два пакета. В примечаниях к выпуску указано: «Красота и суть React не имеет ничего общего с браузерами или DOM... Это [разбиение на два пакета] открывает путь к написанию компонентов, которые могут совместно использоваться веб-версиями React и React Native». Вместо выстраивания предположений, что React станет отображаться только в браузере, будущие выпуски будут нацелены на поддержку отображения на различных платформах.

Нам также понадобится элемент HTML, который будет использоваться ReactDOM для отображения пользовательского интерфейса. В примере 4.1 показано, как добавляются сценарии и элементы HTML. Обе библиотеки доступны в качестве сценариев из Facebook CDN.

Пример 4.1. Настройка HTML-документа с использованием React

```

<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>Pure React Samples</title>
</head>
<body>

  <!-- Целевой контейнер -->
  <div class="react-container"></div>

  <!-- Библиотеки React и ReactDOM -->
  <script src="https://unpkg.com/react@15.4.2/dist/react.js"></script>
  <script src="https://unpkg.com/react-dom@15.4.2/dist/react-dom.js"></script>

  <script>
    // Чистый код React и JavaScript
  </script>

</body>
</html>

```

Это минимальные требования для работы с React в браузере. Можно поместить ваш код JavaScript в отдельный файл, но он должен загружаться на страницу уже после загрузки React.

Виртуальная DOM

HTML — просто набор инструкций, которым следует браузер при построении объектной модели документа (DOM). Когда браузер загружает код HTML и отображает пользовательский интерфейс, элементы, составляющие HTML-документ, становятся элементами DOM.

Предположим, что нужно создать иерархию элементов HTML для кулинарного рецепта. Возможное решение данной задачи может иметь следующий вид:

```

<section id="baked-salmon">
  <h1>Baked Salmon</h1>
  <ul class="ingredients">
    <li>1 lb Salmon</li>
    <li>1 cup Pine Nuts</li>
    <li>2 cups Butter Lettuce</li>
    <li>1 Yellow Squash</li>
    <li>1/2 cup Olive Oil</li>
  </ul>

```

```
    <li>3 cloves of Garlic</li>
  </ul>
  <section class="instructions">
    <h2>Cooking Instructions</h2>
    <p>Preheat the oven to 350 degrees.</p>
    <p>Spread the olive oil around a glass baking dish.</p>
    <p>Add the salmon, garlic, and pine nuts to the dish.</p>
    <p>Bake for 15 minutes.</p>
    <p>Add the yellow squash and put back in the oven for 30 mins.</p>
    <p>Remove from oven and let cool for 15 minutes.
    Add the lettuce and serve.</p>
  </section>
</section>
```

В HTML элементы связаны друг с другом в иерархии, напоминающей генеалогическое древо. Можно сказать, что у корневого элемента имеется три дочерних элемента: заголовок, неупорядоченный список ингредиентов и раздел для инструкций.

По сложившейся традиции сайты состоят из независимых HTML-страниц. При просмотре пользователем этих страниц браузер будет запрашивать и загружать различные HTML-документы. С изобретением Ajax появились одностраничные приложения (single-page application, SPA (https://en.wikipedia.org/wiki/Single-page_application)). Поскольку с помощью Ajax браузеры могут запрашивать и загружать небольшие порции данных, теперь все веб-приложения могут работать на одной странице и в вопросах обновления UI полагаться на JavaScript.

В SPA браузер изначально загружает один HTML-документ. Просматривая сайт, пользователи фактически остаются на той же самой странице. В ходе взаимодействия пользователя с приложением JavaScript разбирает старый и выстраивает новый UI. Может создаваться впечатление перехода от одной страницы к другой, но фактически вы остаетесь на той же самой HTML-странице, а вся нелегкая работа возлагается на JavaScript.

API DOM (https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model) представляет собой коллекцию объектов, которыми JavaScript может воспользоваться для взаимодействия в браузере с целью изменения DOM. Если вам уже приходилось пользоваться методами `document.createElement` или `document.appendChild`, значит, вы уже работали с API DOM. Обновлять или изменять отображаемые элементы DOM в JavaScript относительно легко¹. Но процесс вставки новых элементов идет крайне медленно². Это значит следующее: если разработчики тщательно проработают все моменты внесения изменений в пользовательский интерфейс, то получат возможность повысить производительность приложений.

Эффективное управление изменениями DOM с помощью JavaScript может стать очень сложным и затратным по времени. С точки зрения программиста,

¹ *Simon L.* Minimizing Browser Reflow. <https://developers.google.com/speed/articles/reflow>.

² *Luscher S.* Building User Interfaces with Facebook's React. <https://www.youtube.com/watch?v=1OeXsL5mr4g>, Super VanJS, 2013.

проще будет очистить все дочерние элементы конкретно взятого элемента и выстроить их заново, чем оставить такие дочерние элементы на месте и предпринять попытку их оперативного обновления¹. Проблема заключается в том, что при каждом создании нового приложения эффективной работе с API DOM может мешать отсутствие времени или глубоких познаний в JavaScript. Решением этой проблемы является React.

React представляет собой библиотеку, разработанную для обновления за нас DOM браузера. Больше не придется переживать за сложности, сопряженные с созданием высокопроизводительных SPA, поскольку React может сделать все за нас. Используя React, мы не взаимодействуем с API DOM напрямую. Вместо этого мы имеем дело с виртуальной DOM или с набором инструкций, применяемых React для построения пользовательского интерфейса и организации взаимодействия с браузером².

Виртуальная DOM составлена из элементов React, которые концептуально очень похожи на элементы HTML, но фактически являются объектами JavaScript. Работать непосредственно с последними гораздо быстрее, чем работать с API DOM. Мы вносим изменения в объект JavaScript, в виртуальную DOM, и React отображает эти изменения для нас, используя API DOM наиболее эффективным образом.

Элементы React

DOM браузера составлена из элементов данной модели. По аналогии с этим DOM библиотеки React составлена из элементов React. Элементы DOM и элементы React могут выглядеть одинаково, но на самом деле они совершенно разные. Последние являются описанием того, как должен выглядеть настоящий элемент DOM. Иными словами, элементы React представляют собой инструкции того, как должна быть создана DOM браузера.

Элемент React для представления h1 можно создать, применив метод `React.createElement`:

```
React.createElement("h1", null, "Baked Salmon")
```

Первый аргумент определяет тип элемента, который мы хотим создать. В данном случае нам нужно создать элемент заголовка первого уровня. Третий аргумент представляет дочерние элементы создаваемого нами элемента, то есть любые узлы, вставляемые между открывающим и закрывающим тегами. Второй аргумент представляет свойства создаваемого элемента. Создаваемый здесь элемент типа h1 не имеет никаких свойств.

¹ *Wilton-Jones M.* Efficient JavaScript. <https://dev.opera.com/articles/efficient-javascript/?page=3#reflow>, Dev. Opera.

² React Docs, Refs and the DOM. <https://facebook.github.io/react/docs/refs-and-the-dom.html>.

В ходе отображения React преобразует этот элемент в настоящий элемент DOM:

```
<h1>Baked Salmon</h1>
```

Когда у элемента имеются атрибуты, они могут быть описаны свойствами. Вот как выглядит пример HTML-тега `h1`, имеющего атрибуты `id` и `data-type`:

```
React.createElement("h1",
  {id: "recipe-0", 'data-type': "title"},
  "Baked Salmon"
)
```

```
<h1 data-reactroot id="recipe-0" data-type="title">Baked Salmon</h1>
```

К новому элементу DOM свойства применяются аналогично: добавляются к тегу как атрибуты, а дочерний текст — как текст внутри элемента. Можно также заметить наличие атрибута `data-reactroot`, показывающего, что это корневой элемент вашего React-компонента (рис. 4.1).

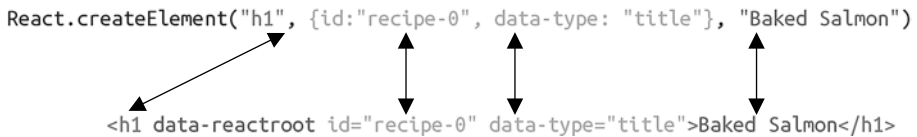


Рис. 4.1. Связь createElement и элемента DOM



Элемент data-reactroot

Элемент `data-reactroot` всегда будет появляться в качестве атрибута корневого элемента вашего React-компонента. До выхода версии 15 к каждому узлу, являющемуся частью вашего компонента, добавлялись идентификаторы React. Это способствовало отображению и отслеживанию того, какой именно элемент нуждается в обновлении. Теперь атрибут добавляется только к корневому элементу, а отображение отслеживает элементы на основе их иерархии.

Следовательно, элемент React представляет собой обычный литерал JavaScript, сообщающий React, как нужно сконструировать элемент DOM. В примере 4.2 показан элемент, фактически создаваемый методом `createElement`.

Пример 4.2. Запись элемента заголовка

```
{
  $$typeof: Symbol(React.element),
  "type": "h1",
  "key": null,
  "ref": null,
```

```

    "props": {"children": "Baked Salmon"},
    "_owner": null,
    "_store": {}
  }
}

```

Это элемент React. В нем имеются поля, используемые библиотекой React: `_owner`, `_store`, `$$typeof`. Для элементов React важную роль играют также поля `key` и `ref`, но они будут рассматриваться позже, в главе 5. А пока уделим более пристальное внимание полям `type` и `props`, показанным в примере 4.2.

Свойство `type` элемента React сообщает библиотеке React, какой тип элемента HTML или SVG нужно создать. Свойство `props` представляет элементы данных и дочерние элементы, требуемые для конструирования элемента DOM. Свойство `children` предназначено для других вложенных элементов в виде текста.



Создание элементов

Мы заглядываем в объект, возвращенный функцией `React.createElement`. Но создавать элементы подобного вида, набирая литералы вручную, вам никогда не придется. Создавать элементы React нужно всегда с помощью функции `React.createElement` или фабрик, речь о которых пойдет в конце данной главы.

ReactDOM

В библиотеке ReactDOM содержатся инструменты, необходимые для отображения элементов React в браузере. Именно в ReactDOM находится метод `render`, а также методы `renderToString` и `renderToStaticMarkup`, используемые на сервере. Мы обсудим их более подробно в главе 12. В этой библиотеке находятся все инструменты, необходимые для создания кода HTML из виртуальной DOM.

Отобразить элемент React, включая его дочерние элементы, в DOM можно с помощью метода `ReactDOM.render`. Элемент, который нужно отобразить, передается как первый аргумент, а в качестве второго используется целевой узел. В нем должен отобразиться элемент:

```

var dish = React.createElement("h1", null, "Baked Salmon")

ReactDOM.render(dish, document.getElementById('react-container'))

```

Отображение заголовочного элемента в DOM приведет к добавлению элемента с заголовком первого уровня к контейнеру `div` с атрибутом `id`, имеющим значение `react-container`, который уже был определен в нашем коде HTML. В примере 4.3 мы создаем этот контейнер внутри тега `body`.

Пример 4.3. React добавляет элемент `h1` к цели: `react-container`

```
<body>
  <div id="react-container">
    <h1>Baked Salmon</h1>
  </div>
</body>
```

Все функциональные возможности отображения в DOM, имеющиеся в React, были перемещены в библиотеку ReactDOM, поскольку библиотека React может использоваться также для создания нативных приложений. Браузер — всего лишь одна из целей React.

И это все, что вам нужно сделать. Вы создаете элемент, а затем отображаете его в DOM. В следующем разделе мы разберемся с тем, как используется `props.children`.

Дочерние элементы

ReactDOM позволяет отображать в DOM единственный элемент¹. React помечает его как `data-reactroot`. Все остальные элементы React составляют внутри этого единственного элемента за счет использования вложенности элементов.

React отображает дочерние элементы с помощью свойства `props.children`. В предыдущем разделе мы отображали текстовый элемент в качестве дочернего для элемента `h1`, вследствие чего для `props.children` было установлено значение "Baked Salmon". Можно в качестве дочерних отобразить и другие элементы React, создавая тем самым дерево элементов. Именно поэтому мы используем такое понятие, как *дерево компонентов*. У дерева есть один корневого компонент, из которого произрастает множество ветвей.

Рассмотрим в примере 4.4 неупорядоченный список, содержащий ингредиенты.

Пример 4.4. Список ингредиентов

```
<ul>
  <li>1 lb Salmon</li>
  <li>1 cup Pine Nuts</li>
  <li>2 cups Butter Lettuce</li>
  <li>1 Yellow Squash</li>
  <li>1/2 cup Olive Oil</li>
  <li>3 cloves of Garlic</li>
</ul>
```

В данном примере неупорядоченный список является корневым элементом, имеющим шесть дочерних. Этот элемент `ul` и его дочерние элементы могут быть представлены с помощью метода `React.createElement` (пример 4.5).

¹ Rendering Elements. <https://facebook.github.io/react/docs/rendering-elements.html>.

Пример 4.5. Неупорядоченный список в качестве элементов React

```

React.createElement(
  "ul",
  null,
  React.createElement("li", null, "1 lb Salmon"),
  React.createElement("li", null, "1 cup Pine Nuts"),
  React.createElement("li", null, "2 cups Butter Lettuce"),
  React.createElement("li", null, "1 Yellow Squash"),
  React.createElement("li", null, "1/2 cup Olive Oil"),
  React.createElement("li", null, "3 cloves of Garlic")
)

```

Каждый дополнительный аргумент, отправляемый функции `createElement`, является еще одним дочерним элементом. React создает массив из таких элементов и устанавливает его в качестве значения для свойства `props.children`.

Если изучить полученный в результате всего этого элемент React, то будет видна каждая запись списка, представленная элементом React и добавленная к массиву, называемому `props.children`. Сейчас так и сделаем (пример 4.6).

Пример 4.6. Получившийся в результате элемент React

```

{
  "type": "ul",
  "props": {
    "children": [
      { "type": "li", "props": { "children": "1 lb Salmon" } ... },
      { "type": "li", "props": { "children": "1 cup Pine Nuts" } ... },
      { "type": "li", "props": { "children": "2 cups Butter Lettuce" } ... },
      { "type": "li", "props": { "children": "1 Yellow Squash" } ... },
      { "type": "li", "props": { "children": "1/2 cup Olive Oil" } ... },
      { "type": "li", "props": { "children": "3 cloves of Garlic" } ... }
    ]
  }
  ...
}

```

Теперь можно увидеть — каждая запись списка является дочерним элементом. Ранее в этой главе был представлен код HTML для всего рецепта, заложенного в элемент `section`. Чтобы сделать то же самое с помощью React, мы воспользуемся серией вызовов `createElement`, показанной в примере 4.7.

Пример 4.7. Дерево элементов React

```

React.createElement("section", {id: "baked-salmon"},
  React.createElement("h1", null, "Baked Salmon"),
  React.createElement("ul", {"className": "ingredients"},
    React.createElement("li", null, "1 lb Salmon"),
    React.createElement("li", null, "1 cup Pine Nuts"),
    React.createElement("li", null, "2 cups Butter Lettuce"),
    React.createElement("li", null, "1 Yellow Squash"),
    React.createElement("li", null, "1/2 cup Olive Oil"),
    React.createElement("li", null, "3 cloves of Garlic")
  ),
)

```

```

    React.createElement("section", {"className": "instructions"},
      React.createElement("h2", null, "Cooking Instructions"),
      React.createElement("p", null, "Preheat the oven to 350 degrees."),
      React.createElement("p", null,
        "Spread the olive oil around a glass baking dish."),
      React.createElement("p", null, "Add the salmon, garlic, and pine..."),
      React.createElement("p", null, "Bake for 15 minutes."),
      React.createElement("p", null, "Add the yellow squash and put..."),
      React.createElement("p", null, "Remove from oven and let cool for 15 ....")
    )
  )
)

```



className в React

Для каждого элемента, имеющего HTML-атрибут class, для соответствующего свойства используется имя className, а не имя class. Поскольку class является в JavaScript зарезервированным словом, для определения атрибута class HTML-элемента нам приходится применять свойство по имени className.

В этом примере показано, как выглядит чистый код React. В конечном итоге в браузере запускается именно такой. Виртуальная DOM — древовидная структура из элементов React, берущих начало из единственного «корня». Элементы React являются инструкциями, которые библиотека React будет использовать, чтобы создать в браузере пользовательский интерфейс.

Конструирование элементов с данными

Главное преимущество использования React — возможность отделения данных от элементов пользовательского интерфейса. Поскольку React, по сути, является кодом JavaScript, то его логику мы можем добавлять, чтобы помочь построить дерево компонентов React. Например, ингредиенты могут храниться в массиве, который можно отобразить на элементы React.

Вернемся к неупорядоченному списку и подумаем над примером 4.8.

Пример 4.8. Неупорядоченный список

```

React.createElement("ul", {"className": "ingredients"},
  React.createElement("li", null, "1 lb Salmon"),
  React.createElement("li", null, "1 cup Pine Nuts"),
  React.createElement("li", null, "2 cups Butter Lettuce"),
  React.createElement("li", null, "1 Yellow Squash"),
  React.createElement("li", null, "1/2 cup Olive Oil"),
  React.createElement("li", null, "3 cloves of Garlic")
);

```

Данные, использованные в этом списке ингредиентов, могут быть легко представлены с помощью массива JavaScript (пример 4.9).

Пример 4.9. Массив записей

```
var items = [
  "1 lb Salmon",
  "1 cup Pine Nuts",
  "2 cups Butter Lettuce",
  "1 Yellow Squash",
  "1/2 cup Olive Oil",
  "3 cloves of Garlic"
]
```

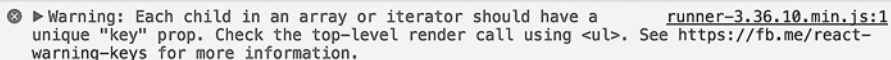
На основе этих данных можно выстроить виртуальную DOM, воспользовавшись, как в примере 4.10, функцией `Array.map`.

Пример 4.10. Отображение элементов массива на элементы `li`

```
React.createElement(
  "ul",
  { className: "ingredients" },
  items.map(ingredient =>
    React.createElement("li", null, ingredient)
  )
)
```

Применение этого синтаксиса приведет к созданию элемента React для каждого имеющегося в массиве ингредиента. Каждая строка показана в дочернем элементе списка в виде текста. Значение для каждого ингредиента показано в виде элемента списка.

При запуске данного кода в консоль будет выведена ошибка, показанная на рис. 4.2.



```
⊗ ▶ Warning: Each child in an array or iterator should have a unique "key" prop. Check the top-level render call using <ul>. See https://fb.me/react-warning-keys for more information. runner-3.36.10.min.js:1
```

Рис. 4.2. Предупреждение, выведенное в консоль

При построении списка из дочерних элементов путем обхода элементов массива библиотеке React нужно, чтобы у каждого из таких элементов было свойство ключа `key`. Это свойство используется библиотекой React с целью помочь эффективному обновлению DOM. Ключи и необходимость их присутствия будут рассматриваться в главе 5, а сейчас вы можете избавиться от данного предупреждения, добавив уникальное свойство `key` к каждому элементу записи списка (пример 4.11). В качестве такого уникального значения может выступать индекс массива для каждого ингредиента.

Пример 4.11. Добавление свойства `key`

```
React.createElement("ul", {className: "ingredients"},
  items.map((ingredient, i) =>
    React.createElement("li", { key: i }, ingredient)
  )
)
```

Компоненты React

Каждый пользовательский интерфейс состоит из частей. В примере используемых здесь кулинарных рецептов имеется несколько, состоящих из частей (рис. 4.3).

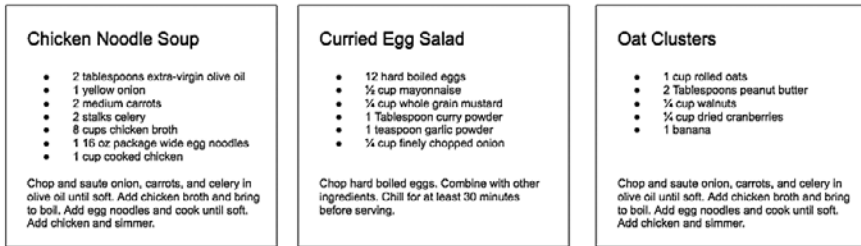


Рис. 4.3. Приложение со списком кулинарных рецептов

В React каждая из этих частей описывается как *компонент*. Компоненты позволяют многократно применять одну и ту же структуру DOM для различных рецептов или наборов данных.

Обдумывая пользовательский интерфейс, создаваемый с помощью React, изыщите возможность разбить ваши элементы на многократно используемые части. Например, у каждого рецепта, показанного на рис. 4.4, имеются заголовок, список ингредиентов и инструкции. Все они являются частью более крупного рецепта или компонента приложения. Компонент можно создать для каждой из выделенных частей: ингредиентов, инструкций и т. д.

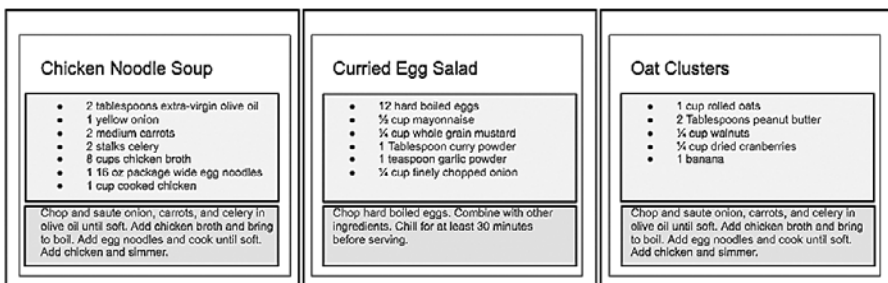


Рис. 4.4. Каждый компонент взят в рамку: App, IngredientsList, Instructions

Подумайте о том, насколько все это поддается масштабированию. Если нужно показать один из рецептов, то наша структура компонентов справится с данной задачей. А если нужно показать 10 000 рецептов, то мы просто создадим новые экземпляры этого компонента.

Изучим три различных способа создания компонентов: `createClass`, классы ES6 и не имеющие состояния функциональные компоненты.

React.createClass

Когда библиотека React была впервые представлена в 2013 году, в ней был только один способ создания компонента: функция `createClass`.

Несмотря на появление новых способов, функция `createClass` по-прежнему широко используется в проектах React. Но команда разработчиков React уже дала сигнал, что в будущем эта функция может быть исключена из перечня рекомендованных средств.

Рассмотрим список ингредиентов, включенных в каждый рецепт. Как показано в примере 4.12, компонент React можно создать с помощью функции `React.createClass`, которая возвращает один элемент неупорядоченного списка, содержащий дочернюю запись списка для каждого ингредиента в массиве.

Пример 4.12. Список ингредиентов в качестве компонента React

```
const IngredientsList = React.createClass({
  displayName: "IngredientsList",
  render() {
    return React.createElement("ul", {"className": "ingredients"},
      React.createElement("li", null, "1 lb Salmon"),
      React.createElement("li", null, "1 cup Pine Nuts"),
      React.createElement("li", null, "2 cups Butter Lettuce"),
      React.createElement("li", null, "1 Yellow Squash"),
      React.createElement("li", null, "1/2 cup Olive Oil"),
      React.createElement("li", null, "3 cloves of Garlic")
    )
  }
})

const list = React.createElement(IngredientsList, null, null)

ReactDOM.render(
  list,
  document.getElementById('react-container')
)
```

Компоненты позволяют задействовать данные для построения многократно используемого UI. В функции `render` для ссылки на экземпляр компонента можно применить ключевое слово `this`, а к свойствам этого экземпляра можно обращаться с помощью `this.props`.

Здесь мы создали элемент, используя наш компонент, и назвали его `IngredientsList`:

```
<IngredientsList>
  <ul className="ingredients">
    <li>1 lb Salmon</li>
```

```

    <li>1 cup Pine Nuts</li>
    <li>2 cups Butter Lettuce</li>
    <li>1 Yellow Squash</li>
    <li>1/2 cup Olive Oil</li>
    <li>3 cloves of Garlic</li>
  </ul>
</IngredientsList>

```

Данные могут передаваться компонентам React в виде свойств. Многократно используемый список ингредиентов можно создать, передавая эти данные списку в виде массива:

```

const IngredientsList = React.createClass({
  displayName: "IngredientsList",
  render() {
    return React.createElement("ul", {className: "ingredients"},
      this.props.items.map((ingredient, i) =>
        React.createElement("li", { key: i }, ingredient)
      )
    )
  }
})

const items = [
  "1 lb Salmon",
  "1 cup Pine Nuts",
  "2 cups Butter Lettuce",
  "1 Yellow Squash",
  "1/2 cup Olive Oil",
  "3 cloves of Garlic"
]

ReactDOM.render(
  React.createElement(IngredientsList, {items}, null),
  document.getElementById('react-container')
)

```

Теперь посмотрим на библиотеку ReactDOM. Свойство данных `items` является массивом с шестью ингредиентами. Поскольку теги `li` создаются с помощью цикла, появляется возможность добавить уникальный ключ с использованием индекса цикла:

```

<IngredientsList items=[...]>
  <ul className="ingredients">
    <li key="0">1 lb Salmon</li>
    <li key="1">1 cup Pine Nuts</li>
    <li key="2">2 cups Butter Lettuce</li>
    <li key="3">1 Yellow Squash</li>
    <li key="4">1/2 cup Olive Oil</li>
    <li key="5">3 cloves of Garlic</li>
  </ul>
</IngredientsList>

```

Компоненты являются объектами. Они, как и классы, могут служить для инкапсуляции кода. Можно создать метод, отображающий одну запись списка, и воспользоваться им для построения всего списка (пример 4.13).

Пример 4.13. Применение специального метода

```
const IngredientsList = React.createClass({
  displayName: "IngredientsList",
  renderListItem(ingredient, i) {
    return React.createElement("li", { key: i }, ingredient)
  },
  render() {
    return React.createElement("ul", {className: "ingredients"},
      this.props.items.map(this.renderListItem)
    )
  }
})
```

Этот фрагмент также соответствует замыслу представлений, используемому в MVC-языках. Все связанное с пользовательским интерфейсом для `IngredientsList` инкапсулировано в один компонент, в котором находится все, что нам нужно.

Теперь можно создать элемент React, используя наш компонент и передавая его списку элементов в качестве свойства. Обратите внимание: теперь тип элемента — строка, которая является непосредственно классом компонента.

**Классы компонента как типы**

При отображении элементов HTML или SVG используются строки. При создании элементов с помощью компонентов класс компонента применяется напрямую. Именно поэтому `IngredientsList` не заключен в кавычки; класс передается `createElement`, поскольку мы имеем дело с компонентом. React создаст экземпляр нашего компонента с этим классом и сделает все за нас.

Использование компонента `IngredientsList` с этими данными приведет к отображению в DOM следующего неупорядоченного списка:

```
<ul data-react-root class="ingredients">
  <li>1 lb Salmon</li>
  <li>1 cup Pine Nuts</li>
  <li>2 cups Butter Lettuce</li>
  <li>1 Yellow Squash</li>
  <li>1/2 cup Olive Oil</li>
  <li>3 cloves of Garlic</li>
</ul>
```

React.Component

Одним из основных свойств, включенных в спецификацию ES6, является `React.Component`, абстрактный класс, который подходит для создания новых компонентов React. Специализированные компоненты могут создаваться путем наследования за счет расширения этого класса с помощью синтаксиса ES6. Задействуя подобный синтаксис, можно создать `IngredientsList` (пример 4.14).

Пример 4.14. IngredientsList в качестве класса ES6

```
class IngredientsList extends React.Component {
  renderListItem(ingredient, i) {
    return React.createElement("li", { key: i }, ingredient)
  }
  render() {
    return React.createElement("ul", {className: "ingredients"},
      this.props.items.map(this.renderListItem)
    )
  }
}
```

Функциональные компоненты, не имеющие состояния

Функциональные компоненты, не имеющие состояния, являются функциями, а не объектами, так что у них нет области видимости `this`. Поскольку это простые, чистые функции, мы будем прибегать к ним в наших приложениях как можно чаще. Может возникнуть ситуация, при которой такой компонент недостаточно надежен, и придется вернуться к применению класса или функции `createClass`, но в общем чем больше используется этот компонент, тем лучше.

Функциональные компоненты, не имеющие состояния, являются функциями, получающими свойства и возвращающими элемент DOM. Эти компоненты хорошо подходят для применения правил функционального программирования. Нужно стремиться сделать каждый такой компонент чистой функцией. Данные компоненты должны получать свойства и возвращать элемент DOM, не вызывая никаких побочных эффектов. Это позволяет добиваться простоты кода и существенно упрощает его тестирование.

Функциональные компоненты, не имеющие состояния, позволят сохранить простоту архитектуры приложения, и команда разработчиков React обещает прирост производительности при использовании этих компонентов. Но если нужно инкапсулировать функциональность или же получить область видимости, соответствующую применению `this`, то применить данные компоненты будет невозможно.

В примере 4.15 функциональность элементов `renderListItem` и `render` объединена в одну функцию.

Пример 4.15. Создание функционального компонента, не имеющего состояния

```
const IngredientsList = props =>
  React.createElement("ul", {className: "ingredients"},
    props.items.map((ingredient, i) =>
      React.createElement("li", { key: i }, ingredient)
    )
  )
```


Отображение приведенного компонента будет осуществляться с помощью метода `ReactDOM.render`, точно таким же способом мы отображали компоненты, созданные с применением `createClass` или синтаксиса класса ES6. Это просто функция. Она собирает данные благодаря аргументам свойств и возвращает неупорядоченный список для каждой записи, отправленной в качестве данных, имеющихся в свойствах.

Один из способов улучшения этого функционального компонента, не имеющего состояния, — деструктуризация аргумента свойств (пример 4.16). Используя синтаксис деструктуризации ES6, можно внести свойство списка непосредственно в область видимости данной функции, сокращая повторяющийся точечный синтаксис. Теперь мы воспользуемся `IngredientsList` точно так же, как делали это при отображении классов компонентов.

Пример 4.16. Деструктуризация аргумента свойств

```
const IngredientsList = ({items}) =>
  React.createElement("ul", {className: "ingredients"},
    items.map((ingredient, i) =>
      React.createElement("li", { key: i }, ingredient)
    )
  )
```



Использование ключевого слова `const` с функциональными компонентами, не имеющими состояния

В каждом из этих компонентов вместо `var` используется ключевое слово `const`. Это устоявшаяся практика, а не непреложное требование. С помощью `const` эта функция объявляется в качестве константы, что удерживает нас от ее последующего переопределения.

Функциональные компоненты, не имеющие состояния, делают синтаксис чище. Кроме того, специалисты Facebook намекнули, что в будущем эти компоненты могут обогнать по быстрдействию код, в котором используется `createClass` или синтаксис класса ES6.

Отображение DOM

Поскольку у нас есть возможность передавать данные нашим компонентам в виде свойств, мы можем отделить данные нашего приложения от логики, используемой для создания пользовательского интерфейса. Тем самым мы получаем изолированный набор данных, с которым намного проще работать, чем с объектной моделью документа. При изменении любого значения из этого изолированного набора данных мы изменяем состояние нашего приложения.

Допустим, все данные нашего приложения хранятся в одном объекте JavaScript. При каждом внесении изменений в данный объект его нужно отправлять компоненту в качестве свойств и заново отображать пользовательский интерфейс. Это значит, что методу `ReactDOM.render` придется потрудиться.

Для того чтобы работа React занимала приемлемый отрезок времени, метод `ReactDOM.render` должен делать свою работу весьма рациональным образом, и он вполне успешно справляется с этой задачей. Вместо опустошения и реконструирования всей DOM метод оставляет текущую модель на месте и применяет минимальный объем изменений, необходимый для трансформации DOM.

Предположим, имеется приложение, показывающее настроение наших пяти сотрудников с помощью либо улыбающегося, либо хмурого лица. Настроение всех пяти сотрудников можно представить в одном JavaScript-массиве данных:

```
["smile", "smile", "frown", "smile", "frown"];
```

Он может использоваться для построения пользовательского интерфейса, который выглядит примерно следующим образом



Если случился аврал и команде пришлось работать все выходные, то новое настроение команды можно отобразить, просто изменив данные в этом массиве. Полученный результат показан на следующем рисунке:

```
["frown", "frown", "frown", "frown", "frown"];
```



Сколько изменений придется внести в первый массив, чтобы он стал похож на второй массив, со всеми хмурыми лицами?

```
["smile", "smile", "frown", "smile", "frown"];
```

```
["frown", "frown", "frown", "frown", "frown"];
```

Нужно изменить первое, второе и четвертое значение со `smile` на `frown`. Таким образом, можно сказать: для изменения данных в первом массиве, чтобы он совпал со вторым массивом, будет внесено три видоизменения.

Теперь рассмотрим способ обновления DOM для отображения этих изменений. Одно неэффективное решение по применению этих изменений к пользовательскому интерфейсу будет заключаться в стирании всей DOM и ее пересоздании, как показано в примере 4.17.

Пример 4.17. Начните с текущего списка

```
<ul>
  <li class="smile">smile</li>
  <li class="smile">smile</li>
  <li class="frown">frown</li>
  <li class="smile">smile</li>
  <li class="frown">frown</li>
</ul>
```

Для выполнения задачи понадобятся следующие шаги.

1. Очистить текущие данные:

```
<ul>
</ul>
```

2. Приступить к цикличному обходу данных и создать первую запись списка:

```
<ul>
  <li class="frown">frown</li>
</ul>
```

3. Создать и добавить вторую запись списка:

```
<ul>
  <li class="frown">frown</li>
  <li class="frown">frown</li>
</ul>
```

4. Создать и добавить третью запись списка:

```
<ul>
  <li class="frown">frown</li>
  <li class="frown">frown</li>
  <li class="frown">frown</li>
</ul>
```

5. Создать и добавить четвертую запись списка:

```
<ul>
  <li class="frown">frown</li>
  <li class="frown">frown</li>
  <li class="frown">frown</li>
  <li class="frown">frown</li>
</ul>
```

6. Создать и добавить пятую запись списка:

```
<ul>
  <li class="frown">frown</li>
  <li class="frown">frown</li>
  <li class="frown">frown</li>
  <li class="frown">frown</li>
  <li class="frown">frown</li>
</ul>
```

Если изменять пользовательский интерфейс путем стирания и пересоздания DOM, то придется создавать и вставлять пять новых элементов модели. Вставка элемента в DOM — одна из самых затратных операций API DOM, которая выполняется довольно медленно. По сравнению с этим обновление элементов, уже находящихся на месте, происходит гораздо быстрее, чем вставка новых.

Метод `ReactDOM.render` вносит изменения, оставляя текущую DOM на месте и просто обновляя те элементы, которые в этом нуждаются. В нашем примере имеются только три видоизменения, так что данный метод должен обновить лишь три элемента DOM (рис. 4.5).

<pre> <li class="smile">smile <li class="smile">smile <li class="frown">frown <li class="smile">smile <li class="frown">frown </pre>	<pre> <li class="frown">frown <li class="frown">frown <li class="frown">frown <li class="frown">frown <li class="frown">frown </pre>
--	--

Рис. 4.5. Обновление трех элементов DOM

Если нужно вставить новые элементы, то библиотека ReactDOM займется этим, но будет стараться свести к минимуму вставки в DOM (являющиеся наиболее затратными операциями).

Такой рациональный подход к отображению DOM необходим для того, чтобы библиотека React выполняла свою работу за приемлемый отрезок времени, поскольку изменения состояния приложения происходят довольно часто. При каждом изменении мы будем полагаться на метод `ReactDOM.render`, чтобы эффективно отобразить обновленный пользовательский интерфейс.

Фабрики

До сих пор единственный способ создания элементов React заключался в использовании метода `React.createElement`. Еще одним способом является применение фабрик. Объект `factory` представляет собой специальный объект, который может служить для абстрагирования от подробностей создания экземпляров объектов. В React фабрики задействуют для того, чтобы помочь создать экземпляры элементов.

В React имеются встроенные фабрики для всех наиболее широко поддерживаемых элементов DOM HTML и SVG, а чтобы создать свои собственные фабрики для конкретных компонентов, можно воспользоваться функцией `React.createFactory`.

Рассмотрим, к примеру, наш элемент `h1`, уже встречавшийся ранее в этой главе:

```
<h1>Baked Salmon</h1>
```

Вместо использования метода `createElement` можно создать элемент React с помощью встроенной фабрики (пример 4.18).

Пример 4.18. Использование функции `createFactory` для создания `h1`

```
React.DOM.h1(null, "Baked Salmon")
```

В данном случае первый аргумент предназначается для свойств, а второй — для дочерних элементов. Как показано в примере 4.19, DOM-фабрики можно также использовать для создания неупорядоченного списка.

Пример 4.19. Создание неупорядоченного списка с помощью DOM-фабрик

```
React.DOM.ul({ "className": "ingredients" },
  React.DOM.li(null, "1 lb Salmon"),
  React.DOM.li(null, "1 cup Pine Nuts"),
  React.DOM.li(null, "2 cups Butter Lettuce"),
  React.DOM.li(null, "1 Yellow Squash"),
  React.DOM.li(null, "1/2 cup Olive Oil"),
  React.DOM.li(null, "3 cloves of Garlic")
)
```

Здесь первый аргумент предназначен для свойств, где мы определяем `className`. Дополнительные аргументы представляют собой элементы, добавляемые к массиву дочерних элементов `children` неупорядоченного списка. Воспользовавшись фабриками, можно также отделить данные ингредиентов и усовершенствовать предыдущее определение (пример 4.20).

Пример 4.20. Использование с фабриками функции `map`

```
var items = [
  "1 lb Salmon",
  "1 cup Pine Nuts",
  "2 cups Butter Lettuce",
  "1 Yellow Squash",
  "1/2 cup Olive Oil",
  "3 cloves of Garlic"
]

var list = React.DOM.ul(
  { className: "ingredients" },
  items.map((ingredient, key) =>
    React.DOM.li({key}, ingredient)
  )
)
ReactDOM.render(
  list,
  document.getElementById('react-container')
)
```

Использование фабрик с компонентами. Если захочется упростить код путем вызова компонентов в качестве функций, то создавать фабрику придется явным образом (пример 4.21).

Пример 4.21. Создание фабрики с использованием `IngredientsList`

```
const { render } = ReactDOM;

const IngredientsList = ({ list }) =>
  React.createElement('ul', null,
    list.map((ingredient, i) =>
      React.createElement('li', {key: i}, ingredient)
    )
  )

const Ingredients = React.createFactory(IngredientsList)

const list = [
  "1 lb Salmon",
  "1 cup Pine Nuts",
  "2 cups Butter Lettuce",
  "1 Yellow Squash",
  "1/2 cup Olive Oil",
  "3 cloves of Garlic"
]

render(
  Ingredients({list}),
  document.getElementById('react-container')
)
```

В данном примере быстрого отображения элемента React можно добиться с помощью фабрики `Ingredients`. Она, как и фабрики DOM, представляет собой функцию, получающую в качестве аргументов свойства и дочерние элементы.

Если вам не приходилось работать с JSX, то фабрики могут показаться более подходящим средством, чем серия вызовов `React.createElement`. Но наиболее простым и распространенным способом определения элементов React является применение JSX-тегов. Если JSX используется с React, то, вполне вероятно, вам уже никогда не придется задействовать фабрики.

В этой главе компоненты React создавались с помощью функций `createElement` и `createFactory`. В главе 5 мы обсудим, как упростить создание компонентов за счет применения JSX.

5 React с JSX

В главе 4 мы выяснили, что виртуальная DOM — набор инструкций, которым React следует при создании и обновлении пользовательского интерфейса. Эти инструкции состоят из объектов JavaScript, называемых элементами React. До сих пор были изучены два способа создания элементов React: через использование функции `React.createElement` и путем применения фабрик.

Альтернативой набору многословных вызовов `React.createElement` является JSX, расширение JavaScript, позволяющее определять элементы React с помощью синтаксиса, похожего на HTML. В этой главе мы обсудим, как применять JSX для конструирования виртуальной DOM с элементами React.

Элементы React в виде кода JSX

Команда Facebook, занимающаяся разработкой React, выпустила JSX одновременно с React, чтобы предоставить лаконичный синтаксис для создания сложных деревьев DOM с атрибутами. Кроме того, разработчики надеялись сделать код React читаемым так же легко, как код HTML и XML.

В JSX тип элемента указывается с помощью тега. Его атрибуты представляют свойства. Дочерние элементы к создаваемому элементу могут добавляться между открывающим и закрывающим тегами.

В качестве дочерних можно добавлять и другие JSX-элементы. При работе с неупорядоченным списком элементы дочерних записей списка добавляются к нему с помощью тегов JSX. Выглядит это очень похоже на HTML (пример 5.1).

Пример 5.1. JSX для неупорядоченного списка

```
<ul>
  <li>1 lb Salmon</li>
```

```
<li>1 cup Pine Nuts</li>
<li>2 cups Butter Lettuce</li>
<li>1 Yellow Squash</li>
<li>1/2 cup Olive Oil</li>
<li>3 cloves of Garlic</li>
</ul>
```

JSX также позволяет работать с компонентами. Нужно просто определить компонент, используя имя класса. На рис. 5.1 массив ингредиентов передается компоненту `IngredientsList` в качестве свойства с помощью JSX.

```
React.createElement(IngredientsList,{list:[...]});
```

```
<IngredientsList list={([...]}>
```

Рис. 5.1. Создание `IngredientsList` с помощью JSX

Когда данному компоненту передается массив ингредиентов, его нужно заключить в фигурные скобки. Это называется *выражением JavaScript* и должно использоваться при передаче значений JavaScript компонентам в качестве свойств. Эти свойства будут получать два типа: либо строку, либо выражение JavaScript. Выражения JavaScript могут содержать массивы, объекты и даже функции. Включить их можно, взяв в фигурные скобки.

Советы по применению JSX

Код JSX может показаться знакомым, и выполнение большинства правил приводит к созданию синтаксиса, похожего на HTML. Тем не менее существует ряд рекомендаций, которые следует усвоить при работе с JSX.

Вложенные компоненты

JSX позволяет добавлять компоненты к другим компонентам в качестве дочерних. Например, внутри `IngredientsList` можно несколько раз отобразить еще один компонент, названный `Ingredient` (пример 5.2).

Пример 5.2. `IngredientsList` с деревом вложенных компонентов `Ingredient`

```
<IngredientsList>
  <Ingredient />
  <Ingredient />
  <Ingredient />
</IngredientsList>
```


className

Поскольку `class` является в JavaScript зарезервированным словом, для определения атрибута `class` вместо него используется `className`:

```
<h1 className="fancy">Baked Salmon</h1>
```

Выражения JavaScript

Выражения JavaScript заключаются в фигурные скобки и показывают, где должны быть вычислены переменные и куда следует вернуть результаты этих вычислений. Например, если нужно отобразить в элементе значение свойства `title`, то можно вставить это значение, используя выражение JavaScript. Переменная будет вычислена, а ее значение возвращено:

```
<h1>{this.props.title}</h1>
```

Значения типов, отличных от строки, должны также фигурировать в качестве выражений JavaScript:

```
<input type="checkbox" defaultChecked={false} />
```

Вычисление

Код JavaScript, добавляемый между фигурными скобками, будет вычислен. Это значит, что будут выполнены такие операции, как объединение или добавление, а также вызваны функции, найденные в выражении JavaScript:

```
<h1>{"Hello" + this.props.title}</h1>

<h1>{this.props.title.toLowerCase().replace}</h1>

function appendTitle({this.props.title}) {
  console.log(`${this.props.title} is great!`)
}
```

Отображение массивов на JSX

JSX — не что иное, как JavaScript, поэтому код JSX можно непосредственно включать в функции JavaScript. Например, отобразить элементы массива на элементы JSX (пример 5.3).

Пример 5.3. `Array.map()` с применением кода JSX

```
<ul>
  {this.props.ingredients.map((ingredient, i) =>
    <li key={i}>{ingredient}</li>
  )}
</ul>
```

Код JSX выглядит понятным и легко читаемым, но не может интерпретироваться браузером. Весь код JSX должен быть преобразован в вызовы функции `createElement` или в вызовы фабрик. И к счастью, для решения этой задачи есть превосходное средство — Babel.

Babel

Большинство языков программирования позволяют компилировать созданный вами исходный код. JavaScript относится к языкам интерпретирующего типа: браузер интерпретирует код, получаемый им в виде текста, поэтому компилировать код JavaScript не нужно. Но не все браузеры поддерживают самый последний синтаксис, описание которого дано в спецификациях ES6 и ES7, и ни один из браузеров не поддерживает синтаксис JSX. Нам хочется воспользоваться самыми новыми свойствами языка JavaScript, а также применить JSX. Так что потребуется способ преобразовать наш причудливый исходный код в то, что браузер сможет интерпретировать. Процесс такого преобразования называется транспиляцией, и именно для его осуществления предназначен Babel (<https://babeljs.io/>).

Первая версия проекта Babel вышла под названием `6to5` в сентябре 2014 года. Она представляла собой инструмент, с помощью которого можно было преобразовать синтаксис ES6 в синтаксис ES5, пользовавшийся более широкой поддержкой браузеров. По мере развития проекта усилия разработчиков были нацелены на создание платформы для поддержки всех самых последних изменений в ECMAScript. Он также дорос и до транспиляции кода JSX в чистый код React. В феврале 2015 года проект был переименован в Babel.

Babel используется в коде, применяемом Facebook, Netflix, PayPal, Airbnb и многими другими проектами. До этого компания Facebook создала JSX-преобразователь, который стал стандартом данной компании, но вскоре от него отказались, отдав предпочтение транспилятору Babel.

Для работы с Babel существует множество способов. Для начала можно воспользоваться самым простым из них, включив ссылку на транспилятор `babel-core` непосредственно в свой HTML, что приведет к транспиляции любого кода в блоках сценария, имеющих тип `text/babel`. Babel транспилирует исходный код на стороне клиента до запуска. Хотя с точки зрения производительности это будет не самым лучшим решением, но для начала работы с JSX ничего лучшего придумать просто невозможно (пример 5.4).

Пример 5.4. Включение `babel-core`

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
```

```
<title>React Examples</title>
</head>
<body>
  <div class="react-container"></div>

  <!-- Библиотеки React и ReactDOM -->
  <script src="https://unpkg.com/react@15.4.2/dist/react.js"></script>
  <script src="https://unpkg.com/react-dom@15.4.2/dist/react-dom.js"></script>
  <script
    src="https://cdnjs.cloudflare.com/ajax/libs/babel-core/5.8.29/browser.js">
  </script>

  <script type="text/babel">
    // Здесь размещается код JSX. Или же ссылка
    // на отдельный файл JavaScript, содержащий код JSX
  </script>

</body>
</html>
```



Вам нужен Babel v5.8

Для транспиляции кода в браузере воспользуйтесь Babel v5.8. Версии Babel 6.0+ в качестве преобразователей, загруженных в браузер, работать не будут.

Чуть позже в этой главе будут рассмотрены способы применения Babel с Webpack, позволяющие производить статическую транспиляцию наших файлов JavaScript. А пока воспользуемся транспилятором, загруженным в браузер.

Применение кода JSX для кулинарных рецептов. Одной из причин нашей растущей привязанности к React является то, что эта библиотека позволяет создавать веб-приложения с красивым кодом. Конструирование хорошо смотрящихся модулей, дающих четкое представление о работе приложения, приносит глубокое внутреннее удовлетворение. JSX представляет весьма привлекательный и конкретный способ выражения элементов React в нашем коде, имеющий для нас вполне определенный смысл, который тут же схватывается при чтении всеми специалистами, входящими в наше сообщество. Недостаток кода JSX заключается в том, что он не читается браузером. Прежде чем браузер сможет интерпретировать наш код, его нужно преобразовать из JSX в чистый React.

Массив в примере 5.5 содержит два рецепта, которые представляют текущее состояние нашего приложения.

Пример 5.5. Массив рецептов

```
var data = [
  {
```

```

    "name": "Baked Salmon",
    "ingredients": [
      { "name": "Salmon", "amount": 1, "measurement": "1 lb" },
      { "name": "Pine Nuts", "amount": 1, "measurement": "cup" },
      { "name": "Butter Lettuce", "amount": 2, "measurement": "cups" },
      { "name": "Yellow Squash", "amount": 1, "measurement": "med" },
      { "name": "Olive Oil", "amount": 0.5, "measurement": "cup" },
      { "name": "Garlic", "amount": 3, "measurement": "cloves" }
    ],
    "steps": [
      "Preheat the oven to 350 degrees.",
      "Spread the olive oil around a glass baking dish.",
      "Add the salmon, garlic, and pine nuts to the dish.",
      "Bake for 15 minutes.",
      "Add the yellow squash and put back in the oven for 30 mins.",
      "Remove from oven and let cool for 15 minutes. Add the lettuce and serve."
    ]
  },
  {
    "name": "Fish Tacos",
    "ingredients": [
      { "name": "Whitefish", "amount": 1, "measurement": "1 lb" },
      { "name": "Cheese", "amount": 1, "measurement": "cup" },
      { "name": "Iceberg Lettuce", "amount": 2, "measurement": "cups" },
      { "name": "Tomatoes", "amount": 2, "measurement": "large" },
      { "name": "Tortillas", "amount": 3, "measurement": "med" }
    ],
    "steps": [
      "Cook the fish on the grill until hot.",
      "Place the fish on the 3 tortillas.",
      "Top them with lettuce, tomatoes, and cheese."
    ]
  }
];

```

Данные выражены в массиве, состоящем из двух объектов JavaScript. Каждый объект содержит название рецепта, список требуемых ингредиентов и список действий, необходимых для приготовления блюда по данному рецепту.

Для этих рецептов можно создать пользовательский интерфейс, состоящий из двух компонентов: `Menu` (Меню), представляющего список рецептов, и `Recipe` (Рецепт), содержащего описание UI для каждого рецепта. Отобразим в DOM компонент `Menu`. Данные ему будут передаваться в виде свойства под названием `recipes` (пример 5.6).

Пример 5.6. Структура кода приложения `Recipes`

```

// Данные, массив объектов приложения Recipes
var data = [ ... ];

// Функциональный компонент, не имеющий состояния, предназначенный
// для отдельно взятого рецепта приложения Recipes
const Recipe = (props) => (
  ...
)

```

```
// Функциональный компонент, не имеющий состояния,
// предназначенный для меню из рецептов
const Menu = (props) => (
  ...
)

// Вызов ReactDOM.render для отображения Menu в текущей DOM-модели
ReactDOM.render(
  <Menu recipes={data} title="Delicious Recipes" />,
  document.getElementById("react-container")
)
```



Поддержка ES6

В файле будет использоваться и код синтаксиса ES6. При транспиляции нашего кода из JSX в чистый React Babel также должен преобразовать код ES6 в пользующийся более широкой поддержкой код ES5 JavaScript, который смогут прочитать практически все браузеры. Особенности всего применяемого здесь кода ES6 были рассмотрены в главе 2.

Элементы React, находящиеся в компоненте Menu, выражены кодом JSX (пример 5.7). Все они помещены в элемент `article`. Для описания DOM нашего меню используются элемент заголовка `h1` и элемент `div.recipes`. Значение для свойства `title` будет показано внутри `h1` в виде текста.

Пример 5.7. Структура компонента Menu

```
const Menu = (props) =>
  <article>
    <header>
      <h1>{props.title}</h1>
    </header>
    <div className="recipes">
      </div>
  </article>
```

В элемент `div.recipes` добавляется компонент для каждого рецепта (пример 5.8).

Пример 5.8. Отображение данных рецептов

```
<div className="recipes">
  {props.recipes.map((recipe, i) =>
    <Recipe key={i} name={recipe.name}
      ingredients={recipe.ingredients}
      steps={recipe.steps} />
  )}
</div>
```

Чтобы вывести список рецептов в элемент `div.recipes`, воспользуемся фигурными скобками, позволяющими добавить выражение JavaScript, возвращающее массив

дочерних элементов. Чтобы вернуть компонент для каждого объекта внутри массива можно в отношении массива `props.recipes` воспользоваться функцией `map`. Как уже упоминалось, в каждом рецепте содержатся название, несколько ингредиентов и инструкции по приготовлению (действия). Эти данные нужно будет передать каждому компоненту `Recipe` в виде свойств. Следует также помнить, что свойство `key` мы используем для присвоения каждому элементу уникального идентификатора.

Код может быть улучшен благодаря оператору распространения JSX. Он работает так же, как и оператор распространения объекта, рассмотренный в главе 2. С его помощью к компоненту `Recipe` будет добавлено каждое поле объекта рецепта. Код, синтаксис которого показан в примере 5.9, выдает те же результаты.

Пример 5.9. Усовершенствование: применение оператора распространения, имеющегося в JSX

```
{props.recipes.map((recipe, i) =>
  <Recipe key={i} {...recipe} />
)}
```

Еще одним местом в нашем компоненте `Menu`, где можно воспользоваться усовершенствованием, предлагаемым синтаксисом ES6, является получение аргумента `props`. Чтобы ввести переменные в область видимости данной функции, можно применить деструктуризацию объекта. Это позволит получить непосредственный доступ к переменным `title` и `recipes`, и тогда перед ними больше не придется указывать префикс `props` (пример 5.10).

Пример 5.10. Реструктуризация компонента `Menu`

```
const Menu = ({ title, recipes }) => (
  <article>
    <header>
      <h1>{title}</h1>
    </header>
    <div className="recipes">
      {recipes.map((recipe, i) =>
        <Recipe key={i} {...recipe} />
      )}
    </div>
  </article>
)
```

Теперь запрограммируем компонент для каждого отдельно взятого рецепта (пример 5.11).

Пример 5.11. Полный код компонента `Recipe`

```
const Recipe = ({ name, ingredients, steps }) =>
  <section id={name.toLowerCase().replace(/ /g, "-")}>
    <h1>{name}</h1>
    <ul className="ingredients">
      {ingredients.map((ingredient, i) =>
```

```

        <li key={i}>{ingredient.name}</li>
      )}
    </ul>
    <section className="instructions">
      <h2>Cooking Instructions</h2>
      {steps.map((step, i) =>
        <p key={i}>{step}</p>
      )}
    </section>
  </section>

```

Этот компонент также относится к функциональным компонентам, не имеющим состояния. У каждого рецепта имеется строка для названия, массив объектов для ингредиентов и массив строк для предпринимаемых действий. Используя имеющуюся в ES6 деструктуризацию объектов, данный компонент можно заставить ввести эти поля по их именам в локальную область видимости. Таким образом мы получим непосредственный доступ к ним без необходимости применять такие формы обращения, как `props.name` или `props.ingredients`, `props.steps`.

Первое показанное здесь выражение JavaScript использовалось для установки атрибута `id` для корневого элемента `section`. Оно преобразует названия рецепта в строку, состоящую из символов нижнего регистра, и повсеместно заменяет в ней пробелы дефисами. В результате, прежде, чем эта строка становится значением атрибута `id` для нашего пользовательского интерфейса, `Baked Salmon` превращается в `baked-salmon` (и аналогично, название рецепта `Boston Baked Beans` будет преобразовано в `boston-baked-beans`). Значение для `name` также отображается в `h1` в качестве текстового узла.

Внутри неупорядоченного списка выражение JavaScript отображает каждый ингредиент в элемент `li`, выводящий название ингредиента. Внутри раздела инструкций можно увидеть точно такой же шаблон, используемый для возвращения элемента абзаца, в котором выводится каждое действие. Эти функции отображения `map` возвращают массивы дочерних элементов.

Полный код приложения должен выглядеть приблизительно так, как показано в примере 5.12.

Пример 5.12. Завершенный код приложения кулинарных рецептов

```

const data = [
  {
    "name": "Baked Salmon",
    "ingredients": [
      { "name": "Salmon", "amount": 1, "measurement": "1 lb" },
      { "name": "Pine Nuts", "amount": 1, "measurement": "cup" },
      { "name": "Butter Lettuce", "amount": 2, "measurement": "cups" },
      { "name": "Yellow Squash", "amount": 1, "measurement": "med" },
      { "name": "Olive Oil", "amount": 0.5, "measurement": "cup" },
    ]
  }
]

```

```

    { "name": "Garlic", "amount": 3, "measurement": "cloves" }
  ],
  "steps": [
    "Preheat the oven to 350 degrees.",
    "Spread the olive oil around a glass baking dish.",
    "Add the salmon, garlic, and pine nuts to the dish.",
    "Bake for 15 minutes.",
    "Add the yellow squash and put back in the oven for 30 mins.",
    "Remove from oven and let cool for 15 minutes. Add the lettuce and serve."
  ]
},
{
  "name": "Fish Tacos",
  "ingredients": [
    { "name": "Whitefish", "amount": 1, "measurement": "1 lb" },
    { "name": "Cheese", "amount": 1, "measurement": "cup" },
    { "name": "Iceberg Lettuce", "amount": 2, "measurement": "cups" },
    { "name": "Tomatoes", "amount": 2, "measurement": "large" },
    { "name": "Tortillas", "amount": 3, "measurement": "med" }
  ],
  "steps": [
    "Cook the fish on the grill until hot.",
    "Place the fish on the 3 tortillas.",
    "Top them with lettuce, tomatoes, and cheese."
  ]
}
]

const Recipe = ({ name, ingredients, steps }) =>
  <section id={name.toLowerCase().replace(/ /g, "-")}>
    <h1>{name}</h1>
    <ul className="ingredients">
      {ingredients.map((ingredient, i) =>
        <li key={i}>{ingredient.name}</li>
      )}
    </ul>
    <section className="instructions">
      <h2>Cooking Instructions</h2>
      {steps.map((step, i) =>
        <p key={i}>{step}</p>
      )}
    </section>
  </section>

const Menu = ({ title, recipes }) =>
  <article>
    <header>
      <h1>{title}</h1>
    </header>
    <div className="recipes">
      {recipes.map((recipe, i) =>
        <Recipe key={i} {...recipe} />
      )}
    </div>
  </article>

```



```
    </div>
  </article>

ReactDOM.render(
  <Menu recipes={data}
    title="Delicious Recipes" />,
  document.getElementById("react-container")
)
```

При запуске этого кода в браузере React выстроит пользовательский интерфейс, показанный на рис. 5.2, используя наши инструкции с данными рецептов.

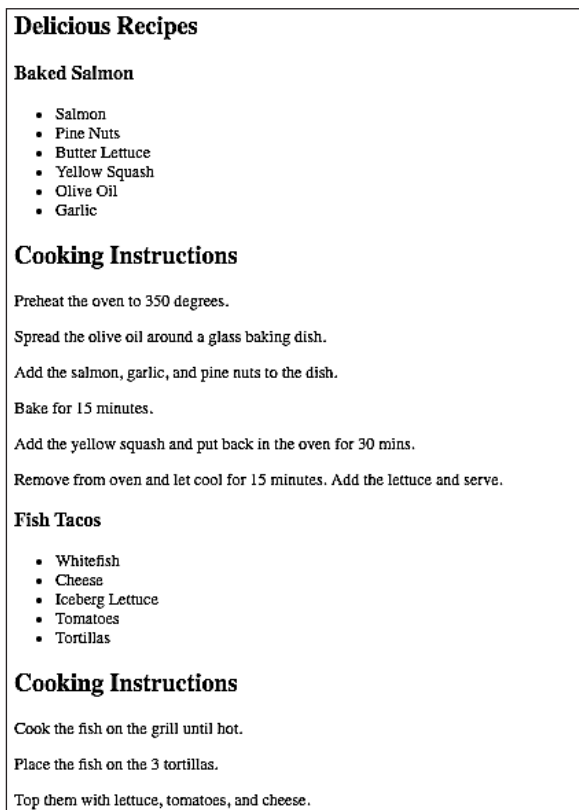


Рис. 5.2. Информация, выводимая на экран приложением Delicious Recipes

Если используется браузер Google Chrome, в котором установлено расширение React Developer Tools Extension, то можно взглянуть на текущее состояние виртуальной DOM. Для этого нужно открыть инструменты разработчика и выбрать вкладку React (рис. 5.3).

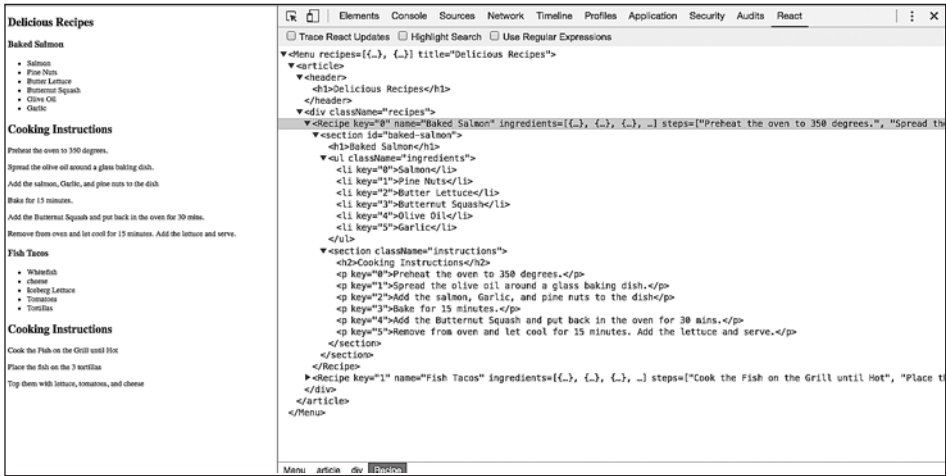


Рис. 5.3. Получающаяся в React Developer Tools виртуальная DOM

Здесь можно увидеть наш компонент `Menu` и его дочерние элементы. Массив `data` включает два объекта для рецептов, и у нас есть два элемента `Recipe`. В каждом из них содержатся свойства для названия рецепта, ингредиентов и действий.

Виртуальная DOM выстроена на основе данных состояния приложения, переданных компоненту `Menu` в качестве свойства. Если внести в массив `recipes` изменения и отобразить наш компонент `Menu` заново, то React изменит эту DOM наиболее эффективным образом.

Заготовки Babel

В Babel 6 возможные преобразования разбиты на модули, названные заготовками (presets). Это требует от специалистов четкого определения запускаемых преобразований путем указания того, какие именно заготовки следует использовать. Цель заключалась в повышении повсеместной модульности, позволяющей разработчикам принимать решение, какой именно синтаксис будет подвергаться преобразованию. Дополнительные модули распределены по нескольким категориям, оптимизированным на основе потребности приложения. Вероятнее всего, вам придется применять следующие заготовки:

- `babel-preset-es2015` — компилирует код ES2015 или ES6 в код ES5;
- `babel-preset-es2016` — компилирует то, что имеется в ES2016, в код ES2015;
- `babel-preset-es2017` — компилирует то, что имеется в ES2017, в код ES2016;
- `babel-preset-env` — компилирует все из кодов ES2015, ES2016, ES2017. Является обобщением всех трех предыдущих заготовок;
- `babel-preset-react` — компилирует код JSX в вызовы `React.createElement`.

Когда в спецификацию ECMAScript предлагают включить новую функцию, она проходит несколько стадий одобрения, от стадии 0, Strawman (только что предложенная и исключительно экспериментальная), до стадии 4, Finished (одобренная в качестве составной части стандарта). Babel предлагает заготовки для каждой из этих стадий, так что вы можете выбирать стадию, которую разрешите в вашем приложении:

- `babel-preset-stage-0`: Strawman;
- `babel-preset-stage-1`: Proposal;
- `babel-preset-stage-2`: Draft;
- `babel-preset-stage-3`: Candidate.

Введение в Webpack

Когда наступает момент практического применения React, возникает масса вопросов. Как справиться с преобразованием JSX и ES6+? Как управлять зависимостями? Как оптимизировать изображения и код CSS?

Чтобы ответить на эти вопросы, было создано множество различных инструментов, включая Browserify, Gulp и Grunt. Благодаря имеющимся функциям и высокой степени востребованности в крупных компаниях, в качестве одного из лидирующих инструментов для создания модулей CommonJS (подробнее о них говорилось в главе 2) стала рассматриваться среда Webpack.

Webpack считается сборщиком (упаковщиком) модулей, получающим все разнообразие ваших файлов (JavaScript, LESS, CSS, JSX, ES6 и т. д.) и превращающим их в один файл. Модульная сборка имеет два основных преимущества: *модульность* и *высокую сетевую производительность*.

Модульность позволяет разбивать исходный код на части, или модули, с которыми проще работать, особенно когда разработка приложения ведется в команде.

Сетевая производительность повышается за счет того, что в браузер требуется загрузить только одну зависимость в виде пакета. Все теги `script` делают HTTP-запросы, каждый из которых влечет крайне невыгодные задержки. Сбор всех зависимостей в один файл позволяет все загружать за один HTTP-запрос, избегая тем самым излишних задержек.

Кроме транспиляции, Webpack может выполнять следующие задачи.

- ❑ *Разбиение кода* — разбиение кода на разные фрагменты, которые могут загружаться по мере необходимости. Иногда они называются свертками или слоями. Цель заключается в том, чтобы разбить код в соответствии с потребностями различных страниц или устройств.

- ❑ *Минификация* — избавление от пробельных символов, разбиений строк, слишком длинных имен переменных и ненужного кода с целью уменьшения размера файла.
- ❑ *Прогон функций (feature flagging)* — отправка кода при тестировании его свойств в одну или несколько сред (но не во все).
- ❑ *Замена модулей без полной перезагрузки (hot module replacement, HMR)* — отслеживание изменения в исходном коде. Немедленная замена только обновленных модулей.

Загрузчики Webpack

Загрузчик представляет собой функцию, управляющую преобразованиями, которым нужно подвергнуть код в ходе сборки. Если в приложении используются код ES6, JSX, CoffeeScript и другие синтаксические усовершенствования языка, которые не могут естественным образом быть прочитаны браузером, то в файле `webpack.config.js` указываются необходимые загрузчики для выполнения работы по преобразованию кода в синтаксис, понятный браузеру.

В Webpack имеется большое количество загрузчиков; их можно разбить на несколько категорий. Чаще всего они используются для транспиляции с одного диалекта в другой. Например, код ES6 и React транспилируется путем включения `babel-loader`. Мы указываем типы файлов, для которых должен быть запущен транспилятор Babel, а все остальные Webpack берет на себя.

Еще одна широко востребованная категория загрузчиков предназначена для стилового оформления. Загрузчик `css-loader` ищет файлы с расширением `.scss` и компилирует их в CSS. Он может использоваться для включения в ваш пакет модулей CSS. Весь код CSS собирается в пакет, как JavaScript, и автоматически добавляется, когда включен связанный с ним файл JavaScript. Применять элементы `link` для включения таблиц стилей не нужно.

Если есть желание просмотреть все варианты, следует ознакомиться с полным списком загрузчиков (<https://webpack.js.org/concepts/loaders/>).

Приложение кулинарных рецептов с применением сборки, выполняемой Webpack

Созданное ранее в данной главе приложение `Recipes` имеет ряд ограничений, которые Webpack поможет смягчить. Использование такого инструмента, как Webpack, для статической сборки кода JavaScript, предназначенного для работы на стороне клиента, позволяет командам совместно разрабатывать крупные веб-приложения. Применяя сборщик пакетов в модули Webpack, можно также получить следующие преимущества.

- ❑ *Модульность* — использование такого модульного шаблона, как CommonJS, для экспорта модулей, которые впоследствии будут импортированы или во-

стребованы другими частями приложения, сделает ваш исходный код более доступным. Это упростит совместную работу команд разработчиков, позволяя им создавать отдельные файлы и работать с ними, а также перед отправкой в виде готовой продукции объединять их в статическом режиме в единый файл.

- ❑ *Возможность составления композиций* — работая с модулями, можно собирать небольшие, простые, многократно применяемые компоненты React, допускающие создание эффективных композиций в рамках приложений. Более мелкие компоненты проще поддаются осмыслению, тестированию и повторному использованию. Их также проще заменять в дальнейшем при расширении приложений.
- ❑ *Скорость* — сборка в единый клиентский пакет всех модулей приложения и зависимостей уменьшит время загрузки вашего приложения за счет сокращения задержек, связанных с каждым HTTP-запросом. Сборка всех составляющих в единый файл означает, что клиенту нужно будет сделать всего один запрос. К сокращению времени загрузки приведет и минификация кода, содержащегося в пакете.
- ❑ *Согласованность* — поскольку сборщик Webpack выполнит транспиляцию кода JSX в код React, а кода ES6 или даже кода ES7 — в универсальный код JavaScript, мы можем приступить к работе, используя завтрашний синтаксис JavaScript уже сегодня. Babel поддерживает широкий диапазон синтаксиса ESNext, что позволяет не волноваться насчет поддержки браузером нашего кода. Таким образом, разработчики получают возможность постоянно применять самые последние новинки синтаксиса JavaScript.

Разбиение компонентов на модули

Подход к разработке приложения Recipes с возможностью использования Webpack и Babel позволяет разбить код на модули, которые задействуют синтаксис ES6. Рассмотрим применяемый для рецептов функциональный компонент, не имеющий состояния (пример 5.13).

Пример 5.13. Текущий компонент Recipe

```
const Recipe = ({ name, ingredients, steps }) =>
  <section id="baked-salmon">
    <h1>{name}</h1>
    <ul className="ingredients">
      {ingredients.map((ingredient, i) =>
        <li key={i}>{ingredient.name}</li>
      )}
    </ul>
    <section className="instructions">
      <h2>Cooking Instructions</h2>
      {steps.map((step, i) =>
        <p key={i}>{step}</p>
      )}
    </section>
  </section>
```

У этого компонента много задач. Он отображает название рецепта, выстраивает неупорядоченный список ингредиентов и показывает инструкции, каждое действие которых получает свой собственный элемент абзаца.

Более функциональный подход к компоненту `Recipe` будет заключаться в его разбиении на меньшие по размеру конкретизированные функциональные компоненты, не имеющие состояния, и в составлении из них композиции. Начать можно с помещения инструкций в их собственные функциональные компоненты, не имеющие состояния, и с создания модуля в отдельном файле, пригодном к использованию для любого набора инструкций (пример 5.14).

Пример 5.14. Компонент `Instructions`

```
const Instructions = ({ title, steps }) =>
  <section className="instructions">
    <h2>{title}</h2>
    {steps.map((s, i) =>
      <p key={i}>{s}</p>
    )}
  </section>

export default Instructions
```

Мы создали новый компонент под названием `Instructions`. Ему будут передаваться заголовок инструкций и действия. Благодаря этому появится возможность его повторного использования для нескольких разновидностей инструкций: `Cooking Instructions`, `Baking Instructions`, `Prep Instructions` или для `Precook Checklist` — то есть для всего, что предполагает действия.

А теперь подумаем об ингредиентах. В компоненте `Recipe` отображаются только названия ингредиентов, но в данных для рецепта у каждого ингредиента есть также количество и единица измерения. Для представления отдельно взятого ингредиента можно создать функциональный компонент, не имеющий состояния (пример 5.15).

Пример 5.15. Компонент `Ingredient`

```
const Ingredient = ({ amount, measurement, name }) =>
  <li>
    <span className="amount">{amount}</span>
    <span className="measurement">{measurement}</span>
    <span className="name">{name}</span>
  </li>

export default Ingredient
```

Предполагается, что у каждого ингредиента имеется количество, единица изменения и название. Мы деструктурируем эти значения из объекта `props` и отобразим каждое из них в независимых элементах `span`, имеющих атрибут `class`.

Используя компонент `Ingredient`, можно выстроить компонент `IngredientsList`, который пригодится всякий раз, когда необходимо отобразить список ингредиентов (пример 5.16).

Пример 5.16. Компонент `IngredientsList`, использующий компонент `Ingredient`

```
import Ingredient from './Ingredient'
const IngredientsList = ({ list }) =>
  <ul className="ingredients">
    {list.map((ingredient, i) =>
      <Ingredient key={i} {...ingredient} />
    )}
  </ul>

export default IngredientsList
```

В этом файле сначала импортируется компонент `Ingredient`, поскольку мы собираемся использовать его для каждого ингредиента. Ингредиенты передаются данному компоненту в виде массива в свойстве под названием `list`. Каждый ингредиент в массиве `list` будет отображен на компонент `Ingredient`. Для передачи всех данных компоненту `Ingredient` в качестве свойств применяется JSX-оператор распространения.

Использование оператора распространения:

```
<Ingredient {...ingredient} />
```

является еще одним способом выражения следующих данных:

```
<Ingredient amount={ingredient.amount}
  measurement={ingredient.measurement}
  name={ingredient.name} />
```

Следовательно, задавая ингредиент с такими полями:

```
let ingredient = {
  amount: 1,
  measurement: 'cup',
  name: 'sugar'
}
```

мы получаем:

```
<Ingredient amount={1}
  measurement="cup"
  name="sugar" />
```

Теперь, имея компоненты для ингредиентов и инструкций, мы можем составить композицию рецептов, используя эти компоненты (пример 5.17).

Пример 5.17. Реструктуризация компонента `Recipe`

```
import IngredientsList from './IngredientsList'
import Instructions from './Instructions'

const Recipe = ({ name, ingredients, steps }) =>
  <section id={name.toLowerCase().replace(/ /g, '-')}>
    <h1>{name}</h1>
```

```

    <IngredientsList list={ingredients} />
    <Instructions title="Cooking Instructions"
      steps={steps} />
  </section>

```

```
export default Recipe
```

Сначала импортируем те компоненты, которыми собираемся воспользоваться, — `IngredientsList` и `Instructions`. Теперь их можно задействовать для введения компонента `Recipe`. Вместо создания пакета из сложного кода, составляющего в одном месте весь рецепт, мы выразили наш рецепт более декларативно путем составления композиции из компонентов меньшего размера. Код стал не только простым и привлекательным, но и хорошо читаемым. В нем показано, что рецепт должен отобразить свое название, список ингредиентов и инструкции по приготовлению блюда. Все касающееся отображения ингредиентов и инструкций было абстрагировано в меньшие по размеру и простые компоненты.

При использовании модульного подхода с применением CommonJS компонент `Menu` выглядит очень похоже. Основное отличие состоит в том, что он будет находиться в своем собственном файле, импортируя требуемые модули и экспортируя самого себя (пример 5.18).

Пример 5.18. Полный код компонента `Menu`

```

import Recipe from './Recipe'

const Menu = ({ recipes }) =>
  <article>
    <header>
      <h1>Delicious Recipes</h1>
    </header>
    <div className="recipes">
      { recipes.map((recipe, i) =>
        <Recipe key={i} {...recipe} />
      ) }
    </div>
  </article>

export default Menu

```

Для отображения компонента `Menu` по-прежнему потребуется библиотека `ReactDOM`. Кроме того, нужен будет файл `index.js`, но выглядеть он будет несколько иначе (пример 5.19).

Пример 5.19. Файл `index.js` в законченном виде

```

import React from 'react'
import { render } from 'react-dom'
import Menu from './components/Menu'
import data from './data/recipes'

```



```
window.React = React
render(
  <Menu recipes={data} />,
  document.getElementById("react-container")
)
```

Первые четыре инструкции предназначены для импортирования модулей, необходимых для работы приложения. Вместо загрузки React и ReactDOM с помощью тега `script` их импортирование выполняется так, чтобы сборщик Webpack смог добавить их к нашему пакету. Нам также понадобятся компонент `Menu` и массив демонстрационных данных, перемещенный в отдельный модуль. В нем, как и прежде, два рецепта: `Baked Salmon` и `Fish Tacos`.

Все импортируемые переменные носят локальный характер для кода файла `index.js`. Присваивание `window.React` значения `React` дает библиотеке React в браузере глобальную область видимости. Тем самым гарантируется работоспособность всех вызовов функции `React.createElement`.

При отображении компонента `Menu` массив с данными рецептов передается этому компоненту в качестве свойства. Вставка и отображение нашего компонента `Menu` производятся одним вызовом `ReactDOM.render`.

Теперь, после разделения кода на отдельные модули и файлы, создадим с помощью Webpack статический процесс сборки, который опять объединит все в один файл.

Установка зависимостей Webpack

Для создания статического процесса сборки с применением Webpack нужно выполнить ряд установок. Все необходимое устанавливается с помощью `npm`. Сначала можно было бы глобально установить сам сборщик для повсеместного использования команды `webpack`:

```
sudo npm install -g webpack
```

Кроме того, Webpack будет работать с Babel, чтобы транспилировать наш код из JSX и ES6 в код JavaScript, запускаемый в браузере. Для выполнения этой задачи мы собираемся использовать несколько загрузчиков, а также ряд заготовок:

```
npm install babel-core babel-loader babel-preset-env babel-preset-react
babel-preset-stage-0 --save-dev
```

В нашем приложении применяются React и ReactDOM. Эти зависимости загружались с помощью тега `script`. Теперь же мы позволим Webpack добавить их в наш пакет. Для этого нужно выполнить локальную установку зависимостей для React и ReactDOM:

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

Тем самым в папку `./node_modules` будут добавлены сценарии, необходимые для `react` и `react-dom`. Вот теперь у нас есть все необходимое для создания процесса статической сборки с помощью Webpack.

Конфигурация Webpack

Чтобы модульное приложение Recipes заработало, нам нужно будет сообщить Webpack, как привязать исходный код к единому файлу. Это можно сделать с помощью конфигурационных файлов, а исходным конфигурационным файлом Webpack неизменно является `webpack.config.js`.

Начальный файл нашего приложения Recipes — `index.js`. Он импортирует React, ReactDOM и файл `Menu.js`. Именно это нам нужно запустить в браузере в первую очередь. Как только сборщику Webpack встречается инструкция `import`, он находит в файловой системе соответствующий модуль и включает его в пакет. Файл `Index.js` импортирует `Menu.js`, `Menu.js` импортирует `Recipe.js`, `Recipe.js` импортирует `Instructions.js` и `IngredientsList.js`, а `IngredientsList.js` импортирует `Ingredient.js`. Webpack последует представленному дереву импортирования и включит в наш пакет все эти необходимые модули.



Инструкция `import` ES6

Мы используем инструкции `import` ES6, которые пока не поддерживаются большинством браузеров или Node.js. Эти инструкции работают благодаря тому, что Babel в финальном коде преобразует их в инструкции `require('модуль/путь');`. Обычно модули CommonJS загружаются с помощью функции `require`.

Поскольку Webpack занимается сборкой нашего пакета, ему нужно указать на необходимость транспиляции JSX в элементы чистого React. Кроме того, нужно преобразовать весь встречающийся синтаксис ES6 в синтаксис ES5. Изначально у нашего процесса сборки будет три этапа (рис. 5.4).

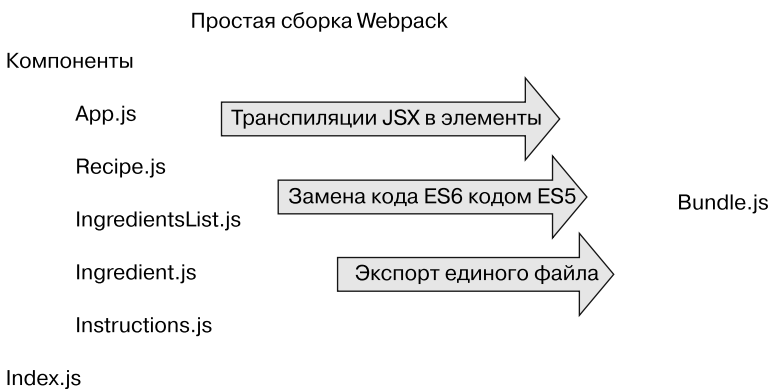


Рис. 5.4. Процесс сборки приложения Recipes

Еще одним модулем, экспортирующим литеральный объект JavaScript с описанием действий, предпринимаемых Webpack, является файл `webpack.config.js` (пример 5.20). Он должен быть сохранен в корневой папке проекта сразу за файлом `index.js`.

Пример 5.20. webpack.config.js

```

module.exports = {
  entry: "./src/index.js",
  output: {
    path: "dist/assets",
    filename: "bundle.js"
  },
  module: {
    rules: [
      {
        test: /\.js$/,
        exclude: /(node_modules)/,
        loader: ['babel-loader'],
        query: {
          presets: ['env', 'stage-0', 'react']
        }
      }
    ]
  }
}

```

Сначала сборщику Webpack указывается, что файлом входа на клиентской стороне будет `./src/index.js`. Он должен автоматически выстроить дерево зависимостей на основе инструкций `import`, начиная с этого файла. Затем указывается, что собранный в пакет файл JavaScript нужно вывести в `./dist/assets/bundle.js`. Именно сюда Webpack поместит окончательно собранный пакет JavaScript.

Следующий набор инструкций для сборщика состоит из списка загрузчиков, запускаемых для указанных модулей. Поле `rules` является массивом, поскольку с помощью Webpack можно включить множество разнообразных загрузчиков. В данном примере включается только `babel`.

Каждый загрузчик — объект JavaScript. Поле `test` представляет собой регулярное выражение, соответствующее путевому имени файла каждого модуля, с которым должен работать загрузчик. В данном случае мы запускаем `babel-loader` для всех импортируемых файлов JavaScript, за исключением находящихся в папке `node_modules`. Начиная работу, `babel-loader` будет использовать заготовки для ES2015 (ES6) и React, чтобы транспилировать синтаксис ES6 или JSX в код JavaScript, запускаемый в большинстве браузеров.

Сборщик Webpack работает в статическом режиме. Как правило, пакеты создаются до развертывания приложения на сервере. Поскольку Webpack был установлен глобально, его можно запустить из командной строки:

```

$ webpack
Time: 1727ms
Asset      Size Chunks          Chunk Names
bundle.js  693 kB    0 [emitted]  main
+ 169 hidden modules

```

Работа Webpack либо завершится успешным созданием пакета, либо даст сбой, при котором покажет сообщение об ошибке. Причиной большинства ошибок являются неправильные ссылки на импортируемые файлы. При отладке Webpack, связанной

с возникновением ошибок, обращайтесь особое внимание на путевые имена файлов, используемые в инструкциях `import`.

Загрузка пакета

Теперь у нас есть пакет, но что с ним делать? Мы экспортируем его в папку `dist`. В ней содержатся файлы, предназначенные для запуска на веб-сервере. Именно в эту папку и нужно поместить файл `index.html` (пример 5.21). В него должен быть включен целевой элемент `div`, в который будет установлен React-компонент `Menu`. В нем также должен быть единственный тег `script`, который загрузит наш помещенный в пакет код JavaScript.

Пример 5.21. `index.html`

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>React Recipes App</title>
</head>
<body>
  <div id="react-container"></div>
  <script src="assets/bundle.js"></script>
</body>
</html>
```

Это главная страница вашего приложения. Она будет загружать все необходимое из одного файла `bundle.js`, используя всего один HTTP-запрос. Данные файлы нужно развернуть на своем веб-сервере или создать приложение веб-сервера, которое будет обслуживать их с помощью таких сред, как Node.js или Ruby on Rails.

Отображение на исходный код

Когда возникает необходимость отладки приложения в браузере, сборка кода в один файл может вызвать ряд проблем, избавиться от которых можно за счет предоставления *отображения на исходный код*. Это файл, отображающий пакет на исходные файлы. Единственное, что нужно сделать для устранения данных проблем при использовании Webpack, — добавить пару строк к нашему файлу `webpack.config.js` (пример 5.22).

Пример 5.22. `webpack.config.js` с отображением на исходный код

```
module.exports = {
  entry: "./src/index.js",
  output: {
    path: "dist/assets",
    filename: "bundle.js",
```

```

    sourceMapFilename: 'bundle.map'
  },
  devtool: '#source-map',
  module: {
    rules: [
      {
        test: /\.js$/,
        exclude: /(node_modules)/,
        loader: ['babel-loader'],
        query: {
          presets: ['env', 'stage-0', 'react']
        }
      }
    ]
  }
}

```

Присваивая свойству `devtool` значения `'#source-map'`, мы указываем сборщику Webpack на необходимость использовать отображение на исходный код. С помощью свойства `sourceMapFilename` нужно указать имя данного файла. Лучше всего называть его именем целевой зависимости. Сборщик свяжет пакет с отображением на исходный код в ходе экспортирования.

Когда Webpack будет запущен еще раз, вы увидите, как будут созданы и добавлены к папке `assets` два выходных файла: исходный `bundle.js` и `bundle.map`.

Отображение на исходный код позволит выполнять отладку, применяя исходные файлы. Во вкладке **Sources** (Источники) инструментов разработчика используемого вами браузера нужно найти папку `webpack://`. Внутри нее вы увидите все исходные файлы, из которых собран пакет (рис. 5.5).

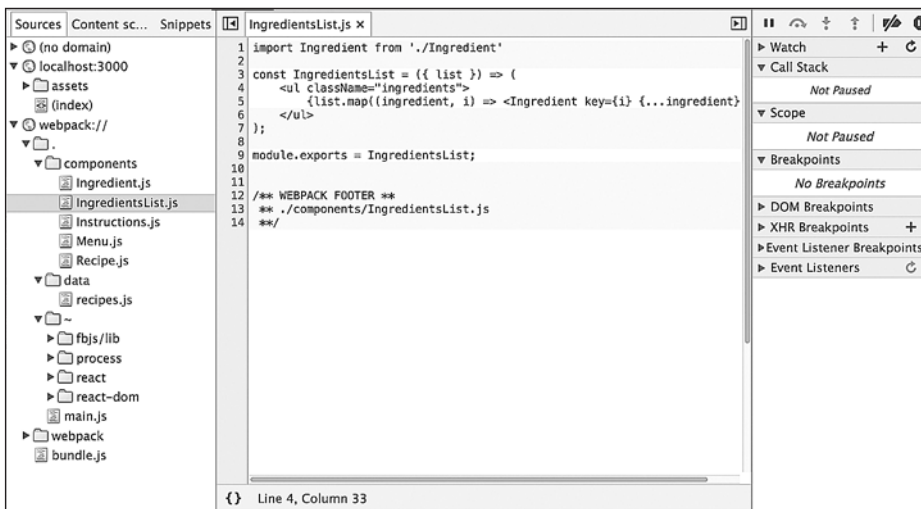


Рис. 5.5. Панель Sources (Источники) инструментов разработчика браузера Chrome

Отладку из этих файлов можно вести с помощью имеющегося в браузере пошагового отладчика. Щелчок кнопкой мыши на любой строке приведет к добавлению контрольной точки. При обновлении содержимого браузера, когда процесс выполнения кода дойдет до места установки любой контрольной точки в вашем исходном файле, обработка кода JavaScript приостановится. Это позволит изучить на панели Scope (Область видимости) переменные, находящиеся в области видимости, или воспользоваться панелью Watch (Отслеживание) и добавить переменные к числу отслеживаемых.

Оптимизация пакета

Получаемый на выходе файл пакета представляет собой простой текстовый файл. Поэтому сокращение содержащегося в нем объема текста приведет к уменьшению размера самого файла, что положительно отразится на скорости его загрузки с помощью протокола HTTP. К числу действий, совершаемых для уменьшения размера файла, относятся удаление всех пробельных символов, урезание имен переменных до одного символа и удаление всех строк кода, до которых интерпретатор никогда не доберется. Уменьшение размера вашего файла JavaScript с применением данных приемов называется *минификацией* кода.

В Webpack имеется встроенный дополнительный модуль, которым можно воспользоваться для минификации пакета. Для этого нужно выполнить локальную установку Webpack:

```
npm install webpack --save-dev
```

Используя дополнительные модули Webpack, к процессу сборки можно добавить дополнительные этапы. В данном примере мы собираемся включить в процесс сборки этап минификации нашего выходного пакета, в результате которого произойдет существенное уменьшение размера файла (пример 5.23).

Пример 5.23. webpack.config.js с дополнительным модулем Uglify

```
var webpack = require("webpack");

module.exports = {
  entry: "./src/index.js",
  output: {
    path: "dist/assets",
    filename: "bundle.js",
    sourceMapFilename: 'bundle.map'
  },
  devtool: '#source-map',
  module: {
    rules: [
      {
        test: /\.js$/,
        exclude: /(node_modules)/,
        loader: ['babel-loader'],
```

```

        query: {
          presets: ['env', 'stage-0', 'react']
        }
      ]
    },
    plugins: [
      new webpack.optimize.UglifyJsPlugin({
        sourceMap: true,
        warnings: false,
        mangle: true
      })
    ]
  }
}

```

Чтобы воспользоваться дополнительным модулем Uglify, нам нужно затребовать сборщик Webpack, для чего понадобится его локальная установка.

Дополнительный модуль `UglifyJsPlugin` представляет собой функцию, получающую инструкции из своих аргументов. После минификации код станет неузнаваемым. Нам понадобится отображение на исходный код, именно поэтому для свойства `sourceMap` установлено значение `true`. Установка для свойства `warnings` значения `false` приведет к удалению из экспортируемого пакета всех предупреждений, выводимых в консоль. Правка имен в нашем коде с установкой в `true` свойства `mangle` означает, что мы собираемся урезать длинные имена переменных, таких как `recipes` или `ingredients`, до одной буквы.

При следующем запуске Webpack станет видно, что размер выходного файла пакета существенно уменьшился, и теперь он стал неузнаваемым. Но включение отображения на исходный код по-прежнему позволит вести отладку из исходного кода, несмотря на проведенную минификацию пакета.

Включение в пакет кода CSS

Еще одним привлекательным свойством сборщика Webpack является его способность включать код CSS в файл с тем же пакетом, в котором находится код JavaScript. Это позволяет пользователям загружать единый файл, содержащий весь необходимый приложению код CSS и JavaScript.

Код CSS может быть включен в пакет с помощью инструкций `import`. Они указывают сборщику на необходимость включить в пакет файлы CSS, связанные с модулем JavaScript:

```

import Recipe from './Recipe'
import '../stylesheets/Menu.css'

const Menu = ({ recipes }) =>
  <article>
    <header>
      <h1>Delicious Recipes</h1>

```

```

    </header>
    <div className="recipes">
      { recipes.map((recipe, i) =>
        <Recipe key={i} {...recipe} />)
      }
    </div>
  </article>

```

```
export default Menu
```

Чтобы в конфигурации Webpack реализовать включение в пакет кода CSS, нужно будет установить ряд загрузчиков:

```
npm install style-loader css-loader postcss-loader --save-dev
```

И наконец, эти загрузчики следует ввести в конфигурацию вашего сборщика Webpack:

```

rules: [
  {
    test: /\.js$/,
    exclude: /(node_modules)/,
    loader: ['babel-loader'],
    query: {
      presets: ['env', 'stage-0', 'react']
    }
  },
  {
    test: /\.css$/,
    use: ['style-loader', 'css-loader', {
      loader: 'postcss-loader',
      options: {
        plugins: () => [require('autoprefixer')]
      }
    }
  ]
]

```

Помещение в пакет файлов CSS с помощью Webpack будет способствовать ускорению загрузки вашего сайта за счет сокращения количества запросов на получение необходимых браузеру ресурсов.

create-react-app

Как говорилось в блоге команды разработчиков Facebook, «экосистема React обычно ассоциируется с потрясающим ростом количества инструментов»¹. В соответствии с этим команда React выпустила запускаемую в командной строке утилиту create-react-app (<https://github.com/facebookincubator/create-react-app>), создающую проект React

¹ Abramov D. Create Apps with No Configuration. <https://facebook.github.io/react/blog/2016/07/22/create-apps-with-no-configuration.html>.

в автоматическом режиме. Эта утилита возникла благодаря проекту Ember CLI (<https://ember-cli.com/>) и позволяет разработчикам осуществлять быстрый запуск нового проекта React без необходимости ручного конфигурирования Webpack, Babel, ESLint и связанных с ними средств.

Чтобы приступить к работе с create-react-app, требуется глобальная установка пакета:

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

Затем нужно воспользоваться одноименной командой с указанием названия папки, в которой нужно создать приложение:

```
create-react-app my-react-project
```

В результате в этой папке будет создан проект React, имеющий всего лишь три зависимости: React, ReactDOM и react-scripts. Инструмент react-scripts, способный творить чудеса, также был создан командой Facebook. Он устанавливает Babel, ESLint, Webpack и другие утилиты, избавляя от необходимости конфигурировать их вручную. В созданной папке проекта находится папка src, содержащая файл App.js. Здесь можно отредактировать корневой компонент и импортировать файлы других компонентов.

Из папки my-react-project можно запустить команду `npm start`. При желании можно также дать команду `yarn start`.

Тесты можно запускать с помощью команд `npm test` или `yarn test`. Это приведет к запуску в интерактивном режиме всех имеющихся в проекте тестовых файлов.

В данном случае ваше приложение будет запущено с использованием порта 3000. Можно также запустить команду `npm run build`. А применяя Yarn, можно дать команду `yarn build`.

В результате будет создан пакет, готовый к практическому применению, уже прошедший транспиляцию и минификацию.

Инструмент create-react-app пригодится как начинающим, так и опытным React-разработчикам. По мере его совершенствования ожидается расширение его возможностей, поэтому стоит почаще заглядывать на GitHub, чтобы следить за изменениями.

6 Свойства, состояние и дерево компонентов

В предыдущей главе речь шла о создании компонентов. Основное внимание при этом уделялось созданию пользовательского интерфейса путем составления композиций из компонентов React. В данной главе будет рассмотрено множество технологий, применяемых для более эффективного управления данными и сокращения времени, затрачиваемого на отладку приложений.

Одним из основных преимуществ работы с React является обработка данных в деревьях компонентов. Методы, которыми можно воспользоваться при этом, позволяют существенно облегчить вашу жизнь в долгосрочной перспективе. Если удастся управлять данными из одного места и создавать пользовательский интерфейс на их основе, то наши приложения будет легче понять и масштабировать.

Проверка свойств

JavaScript относится к языкам со слабой типизацией; это значит, что типы данных у значений переменных могут изменяться. Например, можно сначала установить для переменной JavaScript строковое значение, а затем поменять ее значение на массив, и со стороны языка не будет никаких возражений. Неэффективное управление типами переменных может отрицательно отразиться на времени, затрачиваемом на отладку приложений.

Компоненты React предоставляют способ указания и проверки типов свойств. Использование этой функции существенно сократит время, затрачиваемое на отладку приложений. При предоставлении неверных типов свойств будет выдаваться предупреждение; оно поможет обнаружить ошибки, которые при иных обстоятельствах могут остаться незамеченными.

В React имеется автоматическая встроенная проверка свойств (<https://facebook.github.io/react/docs/typechecking-with-proptypes.html>) на типы переменных, сведения о которой приведены в табл. 6.1.

Таблица 6.1. Проверка свойств React

Тип	Средство проверки
Массивы	React.PropTypes.array
Булевы значения	React.PropTypes.bool
Функции	React.PropTypes.func
Числа	React.PropTypes.number
Объекты	React.PropTypes.object
Строки	React.PropTypes.string

В этом разделе для наших рецептов будет создан компонент `Summary`. Он будет отображать название рецепта и данные о количестве ингредиентов и действий (рис. 6.1).



Рис. 6.1. Информация, выводимая компонентом `Summary` для рецепта `Baked Salmon`

Для отображения этих данных компонент `Summary` нужно снабдить тремя свойствами: названием, массивом ингредиентов и массивом действий. Нам нужно проверить эти свойства, чтобы убедиться: первое из них является строкой, а остальные представляют собой массивы, и предоставить данные по умолчанию на случай их недоступности. Способ реализации проверки свойств зависит от того, как созданы компоненты. Для функциональных компонентов, не имеющих состояния, и для классов ES6 используются разные реализации проверки свойств.

Сначала выясним, зачем нам проверка свойств и как она реализуется в компонентах, созданных с помощью `React.createClass`.

Проверка свойств при использовании `createClass`

Нам нужно выяснить, почему проверке компонентов на допустимость типов придется столь важное значение. Рассмотрим следующую реализацию компонента `Summary`:

```
const Summary = createClass({
  displayName: "Summary",
  render() {
```

```

const {ingredients, steps, title} = this.props
return (
  <div className="summary">
    <h1>{title}</h1>
    <p>
      <span>{ingredients.length} Ingredients</span> |
      <span>{steps.length} Steps</span>
    </p>
  </div>
)
}
})

```

В компоненте `Summary` проводится деструктуризация ингредиентов, действий и названия (`ingredients`, `steps` и `title`) из объекта свойств, после чего создается пользовательский интерфейс для отображения этих данных. Поскольку ожидается, что и ингредиенты, и действия будут представлены в виде массивов, для вычисления количества элементов массива используется метод `Array.length`.

А что произойдет, если этот компонент `Summary` случайно будет выведен на экран с помощью строк?

```

render(
  <Summary title="Peanut Butter and Jelly"
    ingredients="peanut butter, jelly, bread"
    steps="spread peanut butter and jelly between bread" />,
  document.getElementById('react-container')
)

```

Со стороны JavaScript никаких возражений не поступит, но при выяснении длины будет подсчитано количество символов в каждой строке (рис. 6.2).

Peanut Butter and Jelly

27 Ingredients | 44 Steps

Рис. 6.2. Информация, выводимая компонентом `Summary` для рецепта `Peanut Butter and Jelly`

Информация, выводимая этим кодом, выглядит весьма странно. Какими бы ни были составляющие рецепта, весьма сомнительно, что для его реализации понадобятся 27 ингредиентов и 44 действия. Вместо правильного количества действий и ингредиентов мы видим длину каждой строки в символах. Подобную ошибку можно просто не заметить. Но если при создании компонента `Summary` применить проверку, то React отловит эту ошибку за нас:

```

const Summary = createClass({
  displayName: "Summary",
  propTypes: {

```

```

    ingredients: PropTypes.array,
    steps: PropTypes.array,
    title: PropTypes.string
  },
  render() {
    const {ingredients, steps, title} = this.props
    return (
      <div className="summary">
        <h1>{title}</h1>
        <p>
          <span>{ingredients.length} Ingredients | </span>
          <span>{steps.length} Steps</span>
        </p>
      </div>
    )
  }
})

```

Воспользовавшись встроенной в React проверкой типов свойств, мы можем убедиться в том, что и ингредиенты, и действия являются массивами, и, кроме того, удостовериться: название представляет собой строку. Теперь при передаче неверных типов свойств будет выведено сообщение об ошибке (рис. 6.3).

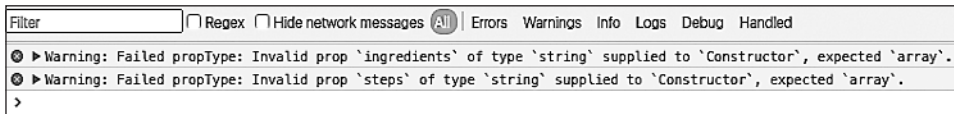


Рис. 6.3. Предупреждение, выдаваемое проверкой типа свойств

А что произойдет, если компонент `Summary` будет отображен без передачи ему каких-либо свойств?

```

render(
  <Summary />,
  document.getElementById('react-container')
)

```

Такое отображение повлечет возникновение ошибки JavaScript, нарушающей работу веб-приложения (рис. 6.4).

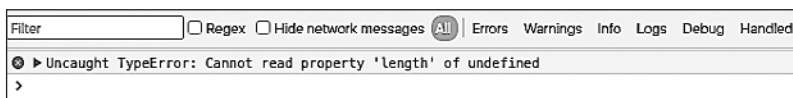


Рис. 6.4. Ошибка, выдаваемая при отсутствии массива

Причина выдачи ошибки заключается в том, что для свойства `ingredients` получен тип `undefined`, который не является объектом, имеющим свойство `length`, как массив или строка. В React есть способ указания требуемых свойств.

Когда такие свойства не предоставлены, библиотека выводит в консоль предупреждение:

```
const Summary = createClass({
  displayName: "Summary",
  propTypes: {
    ingredients: PropTypes.array.isRequired,
    steps: PropTypes.array.isRequired,
    title: PropTypes.string
  },
  render() {
    ...
  }
})
```

Теперь при отображении компонента `Summary` без свойств React обращает наше внимание на эту проблему, выводя предупреждение в консоль непосредственно перед возникновением ошибки. Благодаря этому нам проще будет выяснить причину сбоя (рис. 6.5).

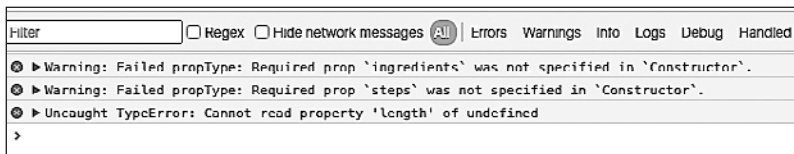


Рис. 6.5. Предупреждения React о непредоставленных свойствах

Компонент `Summary` ожидает предоставления массива для ингредиентов и массива для действий, но он всего лишь использует для каждого массива свойство `length`. Данный компонент разработан для отображения количества элементов в каждом из этих значений. Наверное, резоннее было бы реструктурировать код на ожидание вместо этого числовых значений, поскольку фактически самому компоненту массивы не нужны:

```
import { createClass, PropTypes } from 'react'

export const Summary = createClass({
  displayName: "Summary",
  propTypes: {
    ingredients: PropTypes.number.isRequired,
    steps: PropTypes.number.isRequired,
    title: PropTypes.string
  },
  render() {
    const {ingredients, steps, title} = this.props
    return (
      <div className="summary">
        <h1>{title}</h1>
        <p>
          <span>{ingredients} Ingredients</span> |
```

```

        <span>{steps} Steps</span>
      </p>
    </div>
  )
}
})

```

Использование для этого компонента числовых значений представляется более гибким подходом к решению задачи. Теперь `Summary` просто отображает пользовательский интерфейс; обязанности по фактическому подсчету количества ингредиентов или действий выше по дереву компонентов данный компонент перекладывает на своего родителя или предка.

Свойства, используемые по умолчанию

Еще один способ повышения качества компонентов заключается в присваивании свойствам значений по умолчанию¹. Поведение, проявляемое при проверке, вполне может отвечать вашим ожиданиям: установленные значения по умолчанию будут использоваться, если не предоставлены никакие другие.

Допустим, нужно, чтобы компонент `Summary` работал даже без предоставленных свойств:

```

import { render } from 'react-dom'

render(<Summary />, document.getElementById('react-container'))

```

Используя `createClass`, можно добавить метод `getDefaultProps`, возвращающий значения по умолчанию для тех свойств, которым не присвоены никакие другие значения:

```

const Summary = createClass({
  displayName: "Summary",
  propTypes: {
    ingredients: PropTypes.number,
    steps: PropTypes.number,
    title: PropTypes.string
  },
  getDefaultProps() {
    return {
      ingredients: 0,
      steps: 0,
      title: "[recipe]"
    }
  },
  render() {

```

¹ Документация по React: Default Prop Values. <https://facebook.github.io/react/docs/typechecking-with-proptypes.html#default-prop-values>.

```

const {ingredients, steps, title} = this.props
return (
  <div className="summary">
    <h1>{title}</h1>
    <p>
      <span>{ingredients}</span> Ingredients | </span>
      <span>{steps}</span> Steps</span>
    </p>
  </div>
)
}
}
}

```

Теперь при попытке отобразить этот компонент без свойств вместо него будут показаны данные, установленные по умолчанию (рис. 6.6).



Рис. 6.6. Информация, отображаемая компонентом Summary со свойствами, имеющими значения по умолчанию

Использование свойств со значениями по умолчанию придает гибкость вашему компоненту и исключает возникновение ошибок, когда пользователи не требуют явного предоставления каждого свойства.

Настраиваемая проверка свойств

Встроенные в React средства проверки хорошо подходят для того, чтобы убедиться в правильной востребованности и типизированности ваших переменных. Но есть экземпляры, нуждающиеся в более серьезной проверке. Например, нужно будет убедиться в том, что число находится в указанном диапазоне или значение содержит указанную строку. Для таких случаев React позволяет вам создать собственную проверку.

Настраиваемая проверка реализуется в React с помощью функции. Она должна вернуть либо ошибку, когда не соблюдаются указанные требования по проверке, либо `null`, когда свойство соответствует требованиям.

Используя базовую проверку типов, можно только проанализировать свойство на основании одного условия. А настраиваемая проверка позволяет тестировать свойство многими разнообразными способами. Благодаря применяемой в ней настраиваемой функции сначала убедимся, что значением свойства является строка. Затем наложим на ее длину ограничение в 20 символов (пример 6.1).

Пример 6.1. Настраиваемая проверка свойства

```

propTypes: {
  ingredients: PropTypes.number,
  steps: PropTypes.number,
  title: (props, propName) =>
    (typeof props[propName] !== 'string') ?
      new Error("A title must be a string") :
      (props[propName].length > 20) ?
        new Error(`title is over 20 characters`) :
        null
}

```

Все средства проверки типов свойств являются функциями. Для реализации нашей настраиваемой проверки введем значение свойства `title` объекта `propTypes` в функцию обратного вызова. При отображении компонента `React` внедрит объект `props` и имя текущего свойства в функцию в качестве аргументов. Эти аргументы будут использоваться для проверки конкретного значения указанного свойства.

В данном случае сначала убедимся, является ли свойство `title` строкой. Если нет, то проверка выдаст новую ошибку с сообщением `A title must be a string` (Заголовок должен быть строкой). Если же да, то установим, превышает ли значение этого свойства длину 20 символов. Если нет, то функция возвратит `null`. В противном случае выдаст ошибку. `React` отловит возвращенное сообщение об ошибке и выведет его на консоль в качестве предупреждения.

Настраиваемые проверки позволяют проанализировать конкретные критерии. Такая функция может выполнять несколько проверок и возвращать сообщения об ошибках только при несоблюдении проверяемых условий. Настраиваемые функции — весьма эффективный способ предотвращения ошибок при многократном использовании компонентов.

Классы ES6 и функциональные компоненты, не имеющие состояния

В предыдущих разделах выяснилось, что с помощью `React.createClass` к нашим классам компонентов могут добавляться проверка свойств и значения свойств, используемые по умолчанию. Эта проверка типов также работает для классов ES6 и для функциональных компонентов, не имеющих состояния, но синтаксис, применяемый для ее реализации, немного отличается.

При работе с классами ES6 объявления `propTypes` и `defaultProps` определяются в экземпляре класса, за пределами тела класса. После этого можно установить литералы объектов `propTypes` и `defaultProps` (пример 6.2).

Пример 6.2. Класс ES6

```

class Summary extends React.Component {
  render() {
    const {ingredients, steps, title} = this.props
    return (
      <div className="summary">
        <h1>{title}</h1>
        <p>
          <span>{ingredients} Ingredients | </span>
          <span>{steps} Steps</span>
        </p>
      </div>
    )
  }
}

Summary.propTypes = {
  ingredients: PropTypes.number,
  steps: PropTypes.number,
  title: (props, propName) =>
    (typeof props[propName] !== 'string') ?
      new Error("A title must be a string") :
      (props[propName].length > 20) ?
        new Error(`title is over 20 characters`) :
        null
}

Summary.defaultProps = {
  ingredients: 0,
  steps: 0,
  title: "[recipe]"
}

```

Литералы объектов `propTypes` и `defaultProps` могут также добавляться к функциональным компонентам, не имеющим состояния (пример 6.3).

Пример 6.3. Функциональный компонент, не имеющий состояния

```

const Summary = ({ ingredients, steps, title }) => {
  return <div>
    <h1>{title}</h1>
    <p>{ingredients} Ingredients | {steps} Steps</p>
  </div>
}

Summary.propTypes = {
  ingredients: React.PropTypes.number.isRequired,
  steps: React.PropTypes.number.isRequired
}

Summary.defaultProps = {
  ingredients: 1,
  steps: 1
}

```

Кроме того, с помощью функционального компонента, не имеющего состояния, можно установить для свойств значения, применяемые по умолчанию, непосредственно

в аргументы функции. Для `ingredients`, `steps` и `title` это делается при деструктуризации объекта свойств в аргументах функции:

```
const Summary = ({ ingredients=0, steps=0, title='[recipe]' }) => {
  return <div>
    <h1>{title}</h1>
    <p>{ingredients} Ingredients | {steps} Steps</p>
  </div>
}
```

Статические свойства класса

В предыдущем разделе были показаны способы определения `defaultProps` и `propTypes` за пределами класса. В последних предложениях по спецификации ECMAScript появилась еще одна альтернативная возможность: *поля классов и статические свойства* (class fields & static properties).

Статические свойства класса позволяют инкапсулировать `propTypes` и `defaultProps` внутри объявления класса. Инициализаторы свойств также предоставляют возможность инкапсуляции и более понятный синтаксис:

```
class Summary extends React.Component {

  static propTypes = {
    ingredients: PropTypes.number,
    steps: PropTypes.number,
    title: (props, propName) =>
      (typeof props[propName] !== 'string') ?
        new Error("A title must be a string") :
        (props[propName].length > 20) ?
          new Error(`title is over 20 characters`) :
          null
  }

  static defaultProps = {
    ingredients: 0,
    steps: 0,
    title: "[recipe]"
  }

  render() {
    const {ingredients, steps, title} = this.props
    return (
      <div className="summary">
        <h1>{title}</h1>
        <p>
          <span>{ingredients} Ingredients | </span>
          <span>{steps} Steps</span>
        </p>
      </div>
    )
  }
}
```

В каждом компоненте должны быть реализованы проверка свойств, настраиваемая проверка свойств и возможность устанавливать для них значения по умолчанию. Это облегчит повторное использование компонента, поскольку любые проблемы, связанные с его свойствами, будут отображены на консоли в виде предупреждений.

Ссылки

Ссылки (refs) представляют собой функциональную возможность, позволяющую компонентам React взаимодействовать с дочерними элементами. Чаще всего используются для взаимодействия с элементами пользовательского интерфейса, собирающими вводимые пользователем данные. Рассмотрим HTML-элемент `form`. Такие элементы изначально отображаются на экране, но пользователи могут с ними взаимодействовать и компонент должен соответствующим образом на это реагировать.

Далее в главе мы рассмотрим работу с приложением, позволяющим пользователям сохранять указанные шестнадцатеричные значения цветов и управлять ими. Это приложение, представляющее собой органайзер цветов, дает пользователям возможность добавлять цвета в список. Такой цвет можно оценить или удалить.

Нам понадобится форма для сбора информации от пользователя о новых цветах. Пользователь может предоставить в соответствующих полях название цвета и его шестнадцатеричное значение. Для сбора этих значений из цветового круга компонент `AddColorForm` отображает на экране HTML с полем ввода текста и полем ввода цвета (пример 6.4).

Пример 6.4. AddColorForm

```
import { Component } from 'react'

class AddColorForm extends Component {
  render() {
    return (
      <form onSubmit={e=>e.preventDefault()}>
        <input type="text"
          placeholder="color title..." required/>
        <input type="color" required/>
        <button>ADD</button>
      </form>
    )
  }
}
```

Компонент `AddColorForm` отображает форму HTML, в которой содержатся три элемента: текстовое поле ввода для названия, поле ввода цвета для шестнадцатеричного значения и кнопка для отправки данных формы. Когда она отправлена,

вызывается функция обработки, игнорирующая выдаваемое по умолчанию событие формы. Тем самым пресекается попытка формы выдать при отправке GET-запрос.

После получения формы следует предоставить способ взаимодействия с ней. В частности, при первой ее отправке нужно собрать новую информацию о цвете и перезапустить поля формы, чтобы пользователь имел возможность добавлять цвета. Использование ссылок позволит обращаться к элементам `title` и `color` и взаимодействовать с ними (пример 6.5).

Пример 6.5. AddColorForm с методом отправки submit

```
import { Component } from 'react'

class AddColorForm extends Component {
  constructor(props) {
    super(props)
    this.submit = this.submit.bind(this)
  }
  submit(e) {
    const { _title, _color } = this.refs
    e.preventDefault();
    alert(`New Color: ${_title.value} ${_color.value}`)
    _title.value = '';
    _color.value = '#000000';
    _title.focus();
  }
  render() {
    return (
      <form onSubmit={this.submit}>
        <input ref="_title"
          type="text"
          placeholder="color title..." required/>
        <input ref="_color"
          type="color" required/>
        <button>ADD</button>
      </form>
    )
  }
}
```

К данному классу компонента, использующему синтаксис ES6, добавим конструктор, поскольку отправка `submit` была перемещена в свою собственную функцию. Работая с классами компонентов, применяющими синтаксис ES6, мы должны привязывать область видимости компонента к любым методам, требующим доступ к этой области видимости с помощью ключевого слова `this`.

Затем путем указания в методе `render` на принадлежащий компоненту метод `submit` была произведена настройка обработчика выдаваемого формой события `onSubmit`. Кроме того, добавим поля ссылок `ref` к тем компонентам, на которые нужно будет ссылаться. Поле `ref` является идентификатором того, что React использует для ссылки элементы DOM. Создание `ref`-атрибутов `_title` и `_color` для каждого

элемента ввода данных `input` означает возможность получения доступа к этим элементам с помощью выражений `this.refs_title` или `this.refs_color`.

Когда пользователь добавляет новое название, выбирает новый цвет и отправляет форму, для обработки события вызывается имеющийся в компоненте метод `submit`. Подавив исходное поведение формы при ее отправке, мы выдаем пользователю предупреждение с повтором тех данных, которые были собраны с помощью ссылок. После того как пользователь его отклонит, ссылки используются еще раз для перезапуска значений формы и установки фокуса ввода на поле названия.



Привязка области видимости `this`

При создании компонентов с помощью метода `React.createClass` привязка области видимости `this` к методам компонента не нужна. Данный метод автоматически привязывает `this` за вас.

Обратный поток данных

Конечно, хорошо, когда есть форма, возвращающая введенные данные вместе с предупреждением, но заработать на таком продукте нереально. Нам же нужно собрать данные, введенные пользователем, и отправить их куда-нибудь на обработку. Это значит, что в конечном итоге вся собранная информация должна возвратиться на сервер, то есть проделать путь, который будет рассмотрен в главе 12. Но сначала нужно собрать данные из компонента формы и передать их дальше.

Устоявшимся решением для сбора данных из компонента `React` является использование *обратного потока данных*¹. Действие похоже на *двустороннюю привязку данных*, которая иногда и фигурирует в описаниях. Для этого привлекается отправка компоненту в виде свойства функции обратного вызова, которую компонент может использовать для передачи данных обратно в качестве аргументов. Это называется обратным потоком данных, поскольку функция отправляется компоненту в качестве свойства, а компонент отправляет данные обратно в качестве аргументов функции.

Предположим, что нужно воспользоваться формой цвета, но, когда пользователь отправляет новый образец, следует собрать эту информацию и записать в консоль.

Можно создать функцию под названием `logColor`, получающую название и цвет в качестве аргументов. Их значения могут быть записаны в консоль. При использовании `AddColorForm` для `onNewColor` просто добавляется функциональное свойство,

¹ *Hunt Pete*. Thinking in React. <https://facebook.github.io/react/docs/thinking-in-react.html#step-5-add-inverse-data-flow>.

которое настраивается на функцию `logColor`. Когда пользователь добавляет новый цвет, функция вызывается, и получается, что она отправлена в качестве свойства:

```
const logColor = (title, color) =>
  console.log(`New Color: ${title} | ${value}`)

<AddColorForm onNewColor={logColor} />
```

Чтобы убедиться в правильном направлении потока данных, вызовем `onNewColor` из `props` с соответствующими данными:

```
submit() {
  const {_title, _color} = this.refs
  this.props.onNewColor(_title.value, _color.value)
  _title.value = ''
  _color.value = '#000000'
  _title.focus()
}
```

В нашем компоненте это значит, что мы заменим вызов метода предупреждения `alert` вызовом `this.props.onNewColor` и передадим новые значения названия и цвета, полученные с помощью ссылок.

Роль компонента `AddColorForm` сводится к сбору данных и к передаче их дальше. А за все, что случится с этими данными, он не отвечает. Теперь этой формой можно воспользоваться для сбора данных о цвете от пользователя и передаче их какому-нибудь другому компоненту или методу для обработки собранных данных:

```
<AddColorForm onNewColor={(title, color) => {
  console.log(`TODO: add new ${title} and ${color} to the list`)
  console.log(`TODO: render UI with new Color`)
}} />
```

По готовности можно собрать информацию из этого компонента и добавить новый цвет к нашему списку.



Дополнительные функциональные свойства

Чтобы сделать двустороннюю привязку данных необязательной, перед попыткой вызова функционального свойства нужно проверить факт его существования. Если в последнем примере функциональное свойство `onNewColor` не предоставить, то это приведет к ошибке JavaScript, поскольку компонент будет пытаться вызвать неопределенное значение.

Подобного исхода можно избежать, предварительно проверив, существует ли свойство функции:

```
if (this.props.onNewColor) {
  this.props.onNewColor(_title.value, _color.value)
}
```

Еще разумнее будет определить функциональное свойство в принадлежащем компоненту `propTypes` и `defaultProps`:

```
AddColorForm.propTypes = {
  onNewColor: PropTypes.func
}

AddColorForm.defaultProps = {
  onNewColor: f=>f
}
```

Теперь, когда у предоставляемого свойства будет тип, отличный от функции, React выдаст возражение. Если свойство `onNewColor` предоставлено не будет, то по умолчанию будет использоваться следующая подставная функция: `f=>f`. Это просто функция-заместитель, возвращающая первый переданный ей аргумент. Хотя она ничего не делает, JavaScript может ее вызвать без выдачи сообщений об ошибках.

Ссылки в функциональных компонентах, не имеющих состояния

Ссылки можно задействовать и в функциональных компонентах, не имеющих состояния. В этих компонентах нет ключевого слова `this`, поэтому применить выражение `this.refs` не получится. Вместо использования строковых атрибутов ссылки будут устанавливаться с помощью функции. Она будет передавать нам экземпляр введенных данных в качестве аргумента. А мы сможем забрать этот экземпляр и сохранить в локальной переменной.

Реструктурируем компонент `AddColorForm` в функциональный, не имеющий состояния:

```
const AddColorForm = ({onNewColor=f=>f}) => {
  let _title, _color
  const submit = e => {
    e.preventDefault()
    onNewColor(_title.value, _color.value)
    _title.value = ''
    _color.value = '#000000'
    _title.focus()
  }
  return (
    <form onSubmit={submit}>
      <input ref={input => _title = input}
        type="text"
        placeholder="color title..." required/>
      <input ref={input => _color = input}
        type="color" required/>
      <button>ADD</button>
    </form>
  )
}
```


В данном компоненте ссылки вместо строковых значений устанавливаются с помощью функции обратного вызова. Она передает экземпляр элемента в качестве аргумента. Этот экземпляр может быть взят и сохранен в локальной переменной вида `_title` или `_color`. Поскольку ссылки сохранены в локальных переменных, к ним будет легко получить доступ при отправке данных формы.

Управление состоянием React

До сих пор свойства использовались только для обработки данных в компонентах React. Свойства имеют неизменяемый характер. После отображения свойства компонента не изменяются. Чтобы изменить пользовательский интерфейс, понадобится другой механизм, способный заново отобразить дерево компонента с новыми свойствами. Состояние React является его неотъемлемой частью, предназначенной для управления данными, которые будут изменяться внутри компонента. Когда состояние приложения меняется, пользовательский интерфейс отображается заново, чтобы отразить эти нововведения.

Пользователи взаимодействуют с приложениями. Они перемещаются по данным, ищут их, фильтруют, выбирают, добавляют, обновляют и удаляют. Когда пользователь работает с приложением, состояние программы изменяется, и эти переменные отображаются для пользователя в UI. Появляются и исчезают экраны и панели меню. Меняется видимое содержимое. Включаются и выключаются индикаторы. В React пользовательский интерфейс отражает состояние приложения.

Состояние может быть выражено в компонентах React с единственным объектом JavaScript. В момент изменения своего состояния компонент отображает новый пользовательский интерфейс, показывающий эти изменения. Что может быть функциональнее? Получая данные, компонент React отображает их в виде UI. На основе их изменения React будет обновлять интерфейс, чтобы отразить эти переменные наиболее рациональным образом.

Посмотрим, как можно встроить состояние в наши компоненты React.

Внедрение состояния компонента

Состояние представляет данные, которые при желании можно изменить внутри компонента. Чтобы показать это, рассмотрим компонент `StarRating` (рис. 6.7).



Рис. 6.7. Компонент `StarRating`

Этот компонент требует два важных фрагмента данных: общее количество звезд для отображения и рейтинг, или количество выделенных звезд.

Нам нужен компонент `Star`, реагирующий на щелчки кнопкой мыши и имеющий свойство `selected`. Для каждой звезды может использоваться функциональный компонент, не имеющий состояния:

```
const Star = ({ selected=false, onClick=f=>f }) =>
  <div className={(selected) ? "star selected" : "star"}
    onClick={onClick}>
  </div>
```

```
Star.propTypes = {
  selected: PropTypes.bool,
  onClick: PropTypes.func
}
```

Каждый элемент звезды `Star` будет состоять из контейнера `div`, включающего атрибут `class` со значением `'star'`. Если звезда выбрана, то к ней дополнительно будет добавлен атрибут `class` со значением `'selected'`. У этого компонента также имеется дополнительное свойство `onClick`. Когда пользователь щелкает кнопкой мыши на контейнере `div` какой-нибудь звезды, вызывается данное свойство. Притом родительский компонент, `StarRating`, будет оповещен о щелчке на компоненте `Star`.

Компонент `Star` является функциональным, не имеющим состояния. Исходя из самого названия его категории, в таком компоненте использовать состояние невозможно. Их предназначение — входить в качестве дочерних в состав более сложных компонентов, имеющих состояние. Чем больше компонентов не имеет состояния, тем лучше.



Звезда в CSS

Для создания и отображения звезды в компоненте `StarRating` используется код CSS. В частности, применяя путь обрезки (`clip path`), мы можем обрезать область нашего контейнера `div` так, что он станет похож на звезду. Путь обрезки представляет собой набор точек, из которого составляется многоугольник (`polygon`):

```
.star {
  cursor: pointer;
  height: 25px;
  width: 25px;
  margin: 2px;
  float: left;
  background-color: grey;
  clip-path: polygon(
    50 % 0 %,
    63 % 38 %,
```

```

        100 % 38 %,
        69 % 59 %,
        82 % 100 %,
        50 % 75 %,
        18 % 100 %,
        31 % 59 %,
        0 % 38 %,
        37 % 38 %
    );
}

.star.selected {
    background-color: red;
}

```

У обычной звезды фоновый цвет серый, а у выбранной — красный.

Теперь, получив компонент `Star`, мы можем воспользоваться им для создания компонента `StarRating`. Из своих свойств компонент `StarRating` станет получать общее количество отображаемых звезд. А рейтинг, значение которого пользователь сможет изменять, будет сохранен в состоянии.

Сначала рассмотрим способ внедрения состояния в компонент, определенный с помощью метода `createClass`:

```

const StarRating = createClass({
  displayName: 'StarRating',
  propTypes: {
    totalStars: PropTypes.number
  },
  getDefaultProps() {
    return {
      totalStars: 5
    }
  },
  getInitialState() {
    return {
      starsSelected: 0
    }
  },
  change(starsSelected) {
    this.setState({starsSelected})
  },
  render() {
    const {totalStars} = this.props
    const {starsSelected} = this.state
    return (
      <div className="star-rating">
        {[...Array(totalStars)].map((n, i) =>
          <Star key={i}
            selected={i<starsSelected}
            onClick={() => this.change(i+1)}

```

```
        />
      )}
    <p>{starsSelected} of {totalStars} stars</p>
  </div>
)
}
})
```

Используя метод `createClass`, состояние можно инициализировать, добавив к конфигурации компонента метод `getInitialState` и возвратив объект JavaScript, который изначально устанавливает для переменной состояния `starsSelected` значение `0`.

При отображении компонента общее количество звезд, `totalStars`, берется из свойств компонента и применяется с целью отобразить указанное количество элементов `Star`. При этом для инициализации нового массива указанной длины, отображаемого на элементы `Star`, с конструктором `Array` используется оператор распространения.

Когда компонент отображается на экране, переменная состояния `starsSelected` деструктурируется из элемента `this.state`. Он используется для отображения рейтинга в виде текста в элементе абзаца, а также для подсчета количества выбранных звезд, выводимых на экран. Каждый элемент `Star` получает свое свойство `selected` путем сравнения своего индекса с количеством выбранных звезд. Если выбрано три звезды, то первые три элемента `Star` установят для своего свойства `selected` значение `true`, а все остальные звезды будут иметь для него значение `false`.

И наконец, когда пользователь щелкает кнопкой мыши на отдельно взятой звезде, индекс конкретно этого элемента `Star` увеличивается на единицу и отправляется функции `change`. Данное значение увеличивается на единицу, поскольку предполагается наличие у первой звезды рейтинга `1`, даже притом, что индекс у нее равен нулю.

Инициализация состояния в классе компонента ES6 немного отличается от аналогичного процесса с использованием метода `createClass`. В этих классах состояние может быть инициализировано в конструкторе:

```
class StarRating extends Component {
  constructor(props) {
    super(props)
    this.state = {
      starsSelected: 0
    }
    this.change = this.change.bind(this)
  }

  change(starsSelected) {
    this.setState({starsSelected})
  }
}
```

```

render() {
  const {totalStars} = this.props
  const {starsSelected} = this.state
  return (
    <div className="star-rating">
      {[...Array(totalStars)].map((n, i) =>
        <Star key={i}
          selected={i<starsSelected}
          onClick={() => this.change(i+1)}
        />
      )}
      <p>{starsSelected} of {totalStars} stars</p>
    </div>
  )
}
}

StarRating.propTypes = {
  totalStars: PropTypes.number
}

StarRating.defaultProps = {
  totalStars: 5
}

```

При установке компонента ES6 вызывается его конструктор со свойствами, внедренными в качестве первого аргумента. В свою очередь, эти свойства отправляются родительскому классу путем вызова метода `super`. В данном случае родительским является класс `React.Component`. Вызов `super` инициализирует экземпляр компонента, а `React.Component` придает этому экземпляру функциональную отделку, включающую управление состоянием. После вызова `super` можно инициализировать переменные состояния нашего компонента.

После инициализации состояние функционирует точно так же, как и в компонентах, созданных с помощью метода `createClass`. Оно может быть изменено только путем вызова метода `this.setState`, который обновляет указанные части объекта состояния. После каждого вызова `setState` вызывается функция отображения `render`, обновляющая пользовательский интерфейс в соответствии с новым состоянием.

Инициализация состояния из свойств

Значения состояния можно инициализировать с помощью поступающих свойств. Применение этой схемы может быть вызвано всего лишь несколькими обстоятельствами. Чаще всего ее задействуют при создании компонента, предназначенного для многократного применения среди приложений в различных деревьях компонентов.

При использовании метода `createClass` весьма приемлемым способом инициализации переменных состояния на основе поступающих свойств будет добавление метода `componentWillMount`. Он вызывается один раз при установке компонента, и из этого метода можно вызвать метод `this.setState()`. В нем также имеется доступ к `this.props`; следовательно, чтобы помочь процессу инициализации состояния, можно воспользоваться значениями из `this.props`:

```
const StarRating = createClass({
  displayName: 'StarRating',
  propTypes: {
    totalStars: PropTypes.number
  },
  getDefaultProps() {
    return {
      totalStars: 5
    }
  },
  getInitialState() {
    return {
      starsSelected: 0
    }
  },
  componentWillMount() {
    const { starsSelected } = this.props
    if (starsSelected) {
      this.setState({starsSelected})
    }
  },
  change(starsSelected) {
    this.setState({starsSelected})
  },
  render() {
    const {totalStars} = this.props
    const {starsSelected} = this.state
    return (
      <div className="star-rating">
        {[...Array(totalStars)].map((n, i) =>
          <Star key={i}
            selected={i<starsSelected}
            onClick={() => this.change(i+1)}
          />
        )}
        <p>{starsSelected} of {totalStars} stars</p>
      </div>
    )
  }
})

render(
  <StarRating totalStars={7} starsSelected={3} />,
  document.getElementById('react-container')
)
```

Метод `componentWillMount` является частью жизненного цикла компонента. Он может использоваться для инициализации состояния на основе значений

свойств в компонентах, созданных с помощью метода `createClass`, или в компонентах класса ES6. Более подробно жизненный цикл компонента будет рассмотрен в следующей главе.

Инициализировать состояние внутри класса компонента ES6 можно более простым способом. Конструктор получает свойства в виде аргумента, поэтому просто воспользуемся аргументом `props`, переданным конструктору:

```
constructor(props) {  
  super(props)  
  this.state = {  
    starsSelected: props.starsSelected || 0  
  }  
  this.change = this.change.bind(this)  
}
```

В большинстве случаев нужно избегать установки переменных состояния из свойств. Подобными схемами стоит пользоваться только в самых крайних случаях. Добиться этого не трудно, поскольку при работе с компонентами React ваша задача заключается в ограничении количества компонентов, имеющих состояние¹.



Обновление свойств компонента

При инициализации переменных состояния из свойств компонента может потребоваться заново проинициализировать состояние компонента, когда эти свойства были изменены родительским компонентом. Для решения данной задачи подойдет метод жизненного цикла `componentWillReceiveProps`. Более подробно этот вопрос и доступные методы жизненного цикла компонента рассматриваются в главе 7.

Состояние внутри дерева компонента

Собственное состояние может быть у всех ваших компонентов, но должно ли оно у них быть? Причина удовольствия, получаемого от использования React, не кроется в охоте за переменными состояниями по всему вашему приложению, а исходит из возможности создания простых в понимании масштабируемых приложений. Самое важное, что можно сделать с целью облегчить понимание вашей программы, — это свести к необходимому минимуму количество компонентов, использующих состояние.

¹ Документация по React: Lifting State Up. <https://facebook.github.io/react/docs/lifting-state-up.html>.

Во многих приложениях React есть возможность сгруппировать все данные состояний в корневом компоненте. Данные состояний можно передать вниз по дереву компонентов через свойства, а вверх по дереву к корневому компоненту — через двустороннюю привязку функций. В результате все состояние вашего приложения в целом будет находиться в одном месте. Часто это называют «единым источником истины»¹.

Далее мы рассмотрим, как создаются уровни презентации, где все состояния хранятся в одном месте, в корневом компоненте.

Новый взгляд на приложение органайзера цветов

Органайзер образцов цвета позволяет пользователям добавлять, называть, оценивать и удалять цвета в видеоизменяемых ими списках. Все состояние органайзера может быть представлено с помощью одного массива:

```
{
  colors: [
    {
      "id": "0175d1f0-a8c6-41bf-8d02-df5734d829a4",
      "title": "ocean at dusk",
      "color": "#00c4e2",
      "rating": 5
    },
    {
      "id": "83c7ba2f-7392-4d7d-9e23-35adbe186046",
      "title": "lawn",
      "color": "#26ac56",
      "rating": 3
    },
    {
      "id": "a11e3995-b0bd-4d58-8c48-5e49ae7f7f23",
      "title": "bright red",
      "color": "#ff0000",
      "rating": 0
    }
  ]
}
```

Из этого массива следует, что нам нужно отобразить три цвета: океана в сумерки (ocean at dusk), зеленого газона (lawn) и ярко-красного оттенка (bright red) (рис. 6.8). Массив предоставляет шестнадцатеричные значения, соответствующие тому или иному цвету, и текущий рейтинг для каждого из них, выводимый на экран. Он также дает возможность уникальной идентификации каждого цвета.

¹ Hudson P. State and the Single Source of Truth. <http://www.hackingwithreact.com/read/1/12/stateand-the-single-source-of-truth>, глава 12 издания Hacking with React.

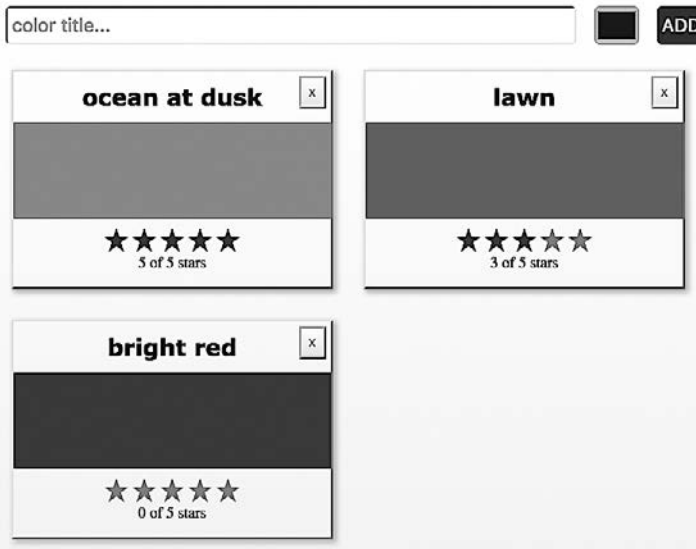


Рис. 6.8. Органайзер с тремя цветами в состоянии

Эти данные состояния будут управлять приложением. Они станут использоваться для создания пользовательского интерфейса при каждом изменении этого объекта. Когда пользователи добавляют или удаляют цвета, эти образцы добавляются в массив или удаляются из него. Когда пользователи дают оценку цветам, рейтинги последних изменяются в массиве.

Передача свойств вниз по дереву компонентов

Ранее в данной главе был создан компонент `StarRating`, сохраняющий рейтинг в состоянии. В органайзере цветов рейтинг сохраняется в каждом объекте цвета. Намного рациональнее будет рассматривать `StarRating` в качестве *презентационного компонента*¹ и объявлять его как функциональный компонент, не имеющий состояния. Презентационные компоненты отвечают только за образ, создаваемый приложением на экране. Они лишь отображают элементы DOM или другие презентационные компоненты. Все данные передаются этим компонентам через свойства, а из них передаются через функции обратного вызова.

Чтобы превратить компонент `StarRating` в чисто презентационный, нужно убрать из него состояние. В презентационных компонентах используются только свойства. Поскольку состояние из этого компонента убирается в момент, когда пользователь

¹ *Abramov D.* Presentational and Container Components. https://medium.com/@dan_abramov/smart-and-dumb-components-7ca2f9a7c7d0.

изменяет оценку, соответствующие данные будут переданы из данного компонента через функцию обратного вызова:

```
const StarRating = ({starsSelected=0, totalStars=5, onRate=f=>f}) =>
  <div className="star-rating">
    {[...Array(totalStars)].map((n, i) =>
      <Star key={i}
        selected={i<starsSelected}
        onClick={() => onRate(i+1)}/>
    )}
    <p>{starsSelected} of {totalStars} stars</p>
  </div>
```

Во-первых, `starsSelected` больше не является переменной состояния, теперь это свойство. Во-вторых, к данному компоненту добавлено свойство в `onRate`, представляющее собой функцию обратного вызова. Вместо того чтобы при изменении пользователем оценки вызвать `setState`, теперь `starsSelected` вызывает `onRate` и отправляет оценку в качестве аргумента.



Состояние в многократно используемых компонентах

Для распространения и многократного использования в различных приложениях может потребоваться создание компонентов пользовательского интерфейса, не имеющих состояния. Совсем не обязательно избавляться абсолютно от всех переменных состояния в компонентах, применяемых исключительно для презентации. Это всего лишь правило, которому рекомендуется следовать, но бывает, что сохранение в презентационном компоненте состояния имеет вполне определенный смысл.

Ограничение, накладываемое на состояние, — размещение его только в одном месте, в корневом компоненте, — означает, что все данные должны передаваться вниз дочерним компонентам в качестве свойств (рис. 6.9).

В органайзере состояние складывается из массива образцов цвета, объявленного в компоненте `App`. Эти цвета передаются вниз компоненту `ColorList` в качестве свойства:

```
class App extends Component {
  constructor(props) {
    super(props)
    this.state = {
      colors: []
    }
  }

  render() {
    const { colors } = this.state
    return (
```

```

    <div className="app">
      <AddColorForm />
      <ColorList colors={colors} />
    </div>
  )
}
}

```

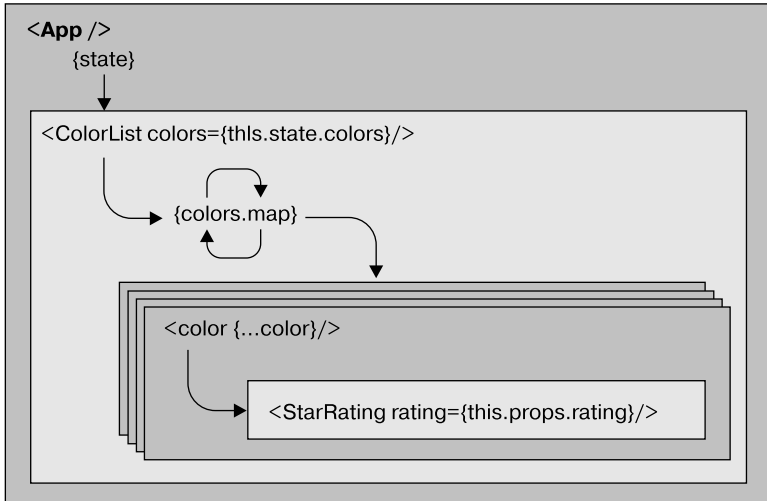


Рис. 6.9. Состояние передается из компонента App дочерним компонентам в качестве свойств

Изначально массив цветов пуст, поэтому компонент `ColorList` вместо каждого цвета будет отображать сообщение. Когда в массиве имеются цвета, данные для каждого отдельно взятого образца передаются компоненту `Color` в качестве свойств:

```

const ColorList = ({ colors=[] }) =>
  <div className="color-list">
    {(colors.length === 0) ?
      <p>No Colors Listed. (Add a Color)</p> :
      colors.map(color =>
        <Color key={color.id} {...color} />
      )
    }
  </div>

```

Теперь компонент `Color` может отобразить название и шестнадцатеричное значение цвета и передать его оценку вниз компоненту `StarRating` в качестве свойства:

```

const Color = ({ title, color, rating=0 }) =>
  <section className="color">
    <h1>{title}</h1>
    <div className="color"
      style={{ backgroundColor: color }}>

```

```

    </div>
    <div>
      <StarRating starsSelected={rating} />
    </div>
  </section>

```

Количество выбранных звезд, `starsSelected`, в звездном рейтинге поступает из оценки каждого цвета. Все данные состояния для каждого цвета переданы вниз по дереву дочерним компонентам в качестве свойств. Когда в корневом компоненте происходят изменения в данных, React, чтобы отразить новое состояние, вносит изменения в пользовательский интерфейс наиболее рациональным образом.

Передача данных вверх по дереву компонентов

Состояние в органайзере цветов может быть обновлено только путем вызова метода `setState` из компонента `App`. Если пользователи инициируют какие-либо изменения из пользовательского интерфейса, то для обновления состояния введенные ими данные нужно будет передать вверх по дереву компонентов компоненту `App` (рис. 6.10). Эта задача может быть выполнена с помощью свойств в виде функций обратного вызова.

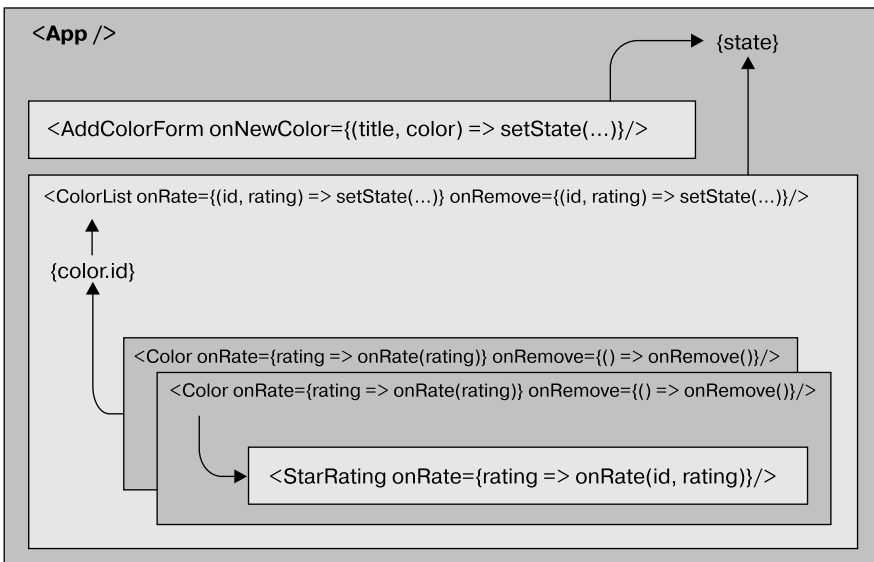


Рис. 6.10. Передача данных вверх корневому компоненту при возникновении событий пользовательского интерфейса

Чтобы добавить новые цвета, нужен способ уникальной идентификации каждого образца. Этот идентификатор будет использоваться для определения местоположения

цвета в массиве состояния. Для создания абсолютно уникальных идентификаторов можно воспользоваться библиотекой `uuid`:

```
npm install uuid --save
```

Все новые цвета будут добавляться в органайзер из компонента `AddColorForm`, который был создан ранее в разделе «Ссылки». У этого компонента имеется дополнительное свойство в виде функции обратного вызова `onNewColor`. Когда пользователь добавляет новый цвет и отправляет данные формы, вызывается функция обратного вызова `onNewColor` с новым названием и шестнадцатеричным значением, полученными от пользователя:

```
import { Component } from 'react'
import { v4 } from 'uuid'
import AddColorForm from './AddColorForm'
import ColorList from './ColorList'

export class App extends Component {

  constructor(props) {
    super(props)
    this.state = {
      colors: []
    }
    this.addColor = this.addColor.bind(this)
  }

  addColor(title, color) {
    const colors = [
      ...this.state.colors,
      {
        id: v4(),
        title,
        color,
        rating: 0
      }
    ]
    this.setState({colors})
  }

  render() {
    const { addColor } = this
    const { colors } = this.state
    return (
      <div className="app">
        <AddColorForm onNewColor={addColor} />
        <ColorList colors={colors} />
      </div>
    )
  }
}
```

Все новые цвета могут быть добавлены из метода `addColor` в компонент `App`. Эта функция привязана к компоненту в конструкторе, следовательно, у нее есть доступ к `this.state` и к `this.setState`.

Новые цвета добавляются путем объединения текущего массива цветов с новым объектом цвета. Идентификатор для последнего устанавливается функцией `v4`, принадлежащей библиотеке `uuid`. Таким образом создается уникальный идентификатор для каждого цвета. Название и цвета передаются методу `addColor` из компонента `AddColorForm`. И наконец, исходным значением для оценки каждого цвета будет ноль.

Когда пользователь добавляет цвет с помощью компонента `AddColorForm`, метод `addColor` обновляет состояние, используя новый список цветов. Как только состояние будет обновлено, компонент `App` заново отобразит дерево компонентов, применяя новый список. После каждого вызова установки состояния `setState` вызывается метод отображения `render`. Новые данные передаются вниз по дереву в качестве свойств и служат для построения пользовательского интерфейса.

Если пользователь хочет оценить или удалить цвет, то нам нужно собрать информацию об этом образце. У каждого цвета будет кнопка удаления: если пользователь нажмет ее, то мы будем знать, что он хочет удалить данный цвет. Кроме того, в случае изменения пользователем оценки цвета с помощью компонента `StarRating` нам нужно изменить рейтинг этого цвета:

```
const Color = ({title,color,rating=0,onRemove=f=>f,onRate=f=>f}) =>
  <section className="color">
    <h1>{title}</h1>
    <button onClick={onRemove}>X</button>
    <div className="color"
      style={{ backgroundColor: color }}>
    </div>
    <div>
      <StarRating starsSelected={rating} onRate={onRate} />
    </div>
  </section>
```

Информация, которая будет изменяться в данном приложении, сохраняется в списке цветов. Поэтому к каждому цвету следует добавить свойства в виде функций обратного вызова `onRemove` и `onRate`, чтобы данные соответствующих событий удаления и оценки передавались вверх по дереву. Свойства в виде функций обратного вызова `onRemove` и `onRate` также будут иметься у компонента `Color`. Когда цвета оцениваются или удаляются, компонент `ColorList` должен оповестить свой родительский компонент `App` о том, что этот цвет нужно оценить или удалить:

```
const ColorList = ({ colors=[], onRate=f=>f, onRemove=f=>f }) =>
  <div className="color-list">
    {(colors.length === 0) ?
      <p>No Colors Listed. (Add a Color)</p> :
      colors.map(color =>
        <Color key={color.id}
          {...color}
          onRate={({rating}) => onRate(color.id, rating)}
          onRemove={() => onRemove(color.id)} />
      )
    }
  </div>
```

Компонент `ColorList` вызовет `onRate`, если какие-либо цвета оценены, или же `onRemove` при удалении каких-либо цветов. Этот компонент управляет коллекцией цветов, отображая их на отдельно взятые компоненты `Color`. Когда отдельно взятые цвета оцениваются или удаляются, `ColorList` идентифицирует цвет, который подвергся воздействию, и передает эту информацию своему родительскому компоненту через свойство в виде функции обратного вызова.

Родительским для `ColorList` является компонент `App`. В нем методы `rateColor` и `removeColor` могут быть добавлены и привязаны к экземпляру компонента в конструкторе. Как только понадобится оценить или удалить цвет, эти методы обновят состояние. Они добавлены к компоненту `ColorList` в качестве свойств в виде функций обратного вызова:

```
class App extends Component {
  constructor(props) {
    super(props)
    this.state = {
      colors: []
    }
    this.addColor = this.addColor.bind(this)
    this.rateColor = this.rateColor.bind(this)
    this.removeColor = this.removeColor.bind(this)
  }

  addColor(title, color) {
    const colors = [
      ...this.state.colors,
      {
        id: v4(),
        title,
        color,
        rating: 0
      }
    ]
    this.setState({colors})
  }

  rateColor(id, rating) {
    const colors = this.state.colors.map(color =>
      (color.id !== id) ?
        color :
        {
          ...color,
          rating
        }
    )
    this.setState({colors})
  }

  removeColor(id) {
    const colors = this.state.colors.filter(
      color => color.id !== id
    )
    this.setState({colors})
  }
}
```

```
render() {
  const { addColor, rateColor, removeColor } = this
  const { colors } = this.state
  return (
    <div className="app">
      <AddColorForm onNewColor={addColor} />
      <ColorList colors={colors}
        onRate={rateColor}
        onRemove={removeColor} />
    </div>
  )
}
```

Оба метода: и `rateColor`, и `removeColor` — ожидают получения идентификатора того цвета, который оценивается или удаляется. ID записывается в компоненте `ColorList` и передается в качестве аргумента методам `rateColor` или `removeColor`. Метод `rateColor` находит оцениваемый цвет и изменяет его рейтинг в состоянии. Для создания нового массива состояния без удаленного цвета метод `removeColor` использует метод `Array.filter`.

После вызова метода `setState` пользовательский интерфейс отображается заново с новыми данными состояния. Все данные, изменившиеся в этом приложении, управляются из одного компонента, `App`. Приведенный подход существенно упрощает понимание того, какие именно данные используются приложением для создания состояния и как они будут изменены.

Компоненты React отличаются высокой надежностью. Они предоставляют четко выраженный способ управления свойствами и их проверки, обмена данными с дочерними элементами и управления данными состояния из-за пределов компонента. Эти свойства дают возможность создать превосходно масштабируемые уровни презентации.

Уже не раз упоминалось, что состояние предназначено для данных, подвергаемых изменениям. Состояние также можно использовать в вашем приложении для кэширования данных. Например, если имеется список записей, в котором пользователь может вести поиск, то этот список можно в ходе поиска сохранить в состоянии.

Часто высказываются рекомендации ограничивать присутствие состояния корневыми компонентами. Такой подход будет встречаться во многих приложениях React. Как только ваше приложение достигнет определенного размера, двусторонняя привязка данных и свойства, передаваемые явным образом, могут принести массу неудобств.

Чтобы управлять состоянием и сократить объем шаблонного кода в подобных ситуациях, можно применять шаблон проектирования Flux и Flux-библиотеки, подобные Redux.

React — относительно небольшая библиотека, и на данный момент мы рассмотрели основную часть ее функциональных возможностей. В следующей главе мы обсудим такие основные свойства компонентов React, как жизненный цикл компонента и компоненты высшего порядка.

7

Усовершенствование компонентов

До сих пор мы рассматривали вопросы установки и составления композиций компонентов для создания с помощью React презентационных уровней приложения. Исключительное использование имеющегося в компонентах React метода `render` позволяет создавать целый ряд приложений. Но мир JavaScript совсем не прост. В нем повсеместно властвует асинхронность. При загрузке данных приходится сталкиваться с задержками. Их же приходится преодолевать при разработке анимаций. Вероятнее всего, чтобы сориентироваться в сложности реального мира JavaScript, предпочтение будет отдано библиотекам этого языка.

Приложения можно совершенствовать с помощью библиотек JavaScript, созданных сторонними разработчиками, или путем запроса внутренних данных. Но прежде нужно понять, как работать с *жизненным циклом компонента*: серией методов, которые могут вызываться при каждой установке или обновлении компонента.

Данную главу мы начнем с исследования жизненного цикла компонента. После представления этого цикла изучим приемы его применения для загрузки данных, внедрения средств JavaScript, созданных сторонними разработчиками, и даже для повышения производительности работы компонентов. Затем рассмотрим вопросы повторного использования функциональных средств во всех наших приложениях с помощью *компонентов высшего порядка*. Кроме того, обсудим альтернативные архитектуры приложений, управляющие состоянием исключительно за пределами React.

Жизненные циклы компонентов

Жизненный цикл компонента состоит из методов, последовательно вызываемых при установке или обновлении компонента. Они вызываются либо перед тем, либо после того, как компонент отобразит пользовательский интерфейс. Фактически

частью жизненного цикла компонента является сам метод отображения `render`. Существует два основных вида цикла: жизненный цикл установки и жизненный цикл обновления.

Жизненный цикл установки

Жизненный цикл установки состоит из методов, вызываемых при установке компонента или при его удалении с экрана. Иными словами, эти методы позволяют изначально устанавливать состояние, совершать вызовы API, запускать и останавливать таймеры, манипулировать отображаемой DOM, инициализировать библиотеки сторонних разработчиков и т. д. Эти методы позволяют вводить в обращение JavaScript, чтобы помочь инициализировать и уничтожать компонент.

Жизненный цикл установки отличается в зависимости от того, как создаются компоненты: с помощью синтаксиса ES6 или метода `React.createClass`. При использовании последнего для получения свойств компонента сначала вызывается метод `getDefaultProps`. Затем для инициализации состояния вызывается метод `getInitialState`.

У классов ES6 нет этих методов. Вместо них получают свойства, используемые по умолчанию, которые в виде аргументов передаются конструктору. Состояние инициализируется в конструкторе. Доступ к свойствам имеется как у конструкторов класса ES6, так и у метода `getInitialState`, и, если нужно, их можно задействовать, чтобы помочь определить исходное состояние.

Методы жизненного цикла установки компонента перечислены в табл. 7.1.

Таблица 7.1. Жизненный цикл установки компонента

Класс ES6	React.createClass()
	<code>getDefaultProps()</code>
<code>constructor(props)</code>	<code>getInitialState()</code>
<code>componentWillMount()</code>	<code>componentWillMount()</code>
<code>render()</code>	<code>render()</code>
<code>componentDidMount()</code>	<code>componentDidMount()</code>
<code>componentWillUnmount()</code>	<code>componentWillUnmount()</code>



Конструкторы класса

С технической точки зрения конструктор не метод жизненного цикла. Он включен в перечень из-за того, что используется для инициализации компонента (при которой инициализируется состояние). Кроме того, конструктор всегда является первой функцией, вызываемой при установке компонента.

После получения свойств и инициализации состояния вызывается метод `componentWillMount`. Он вызывается до отображения DOM и может быть использован для инициализации библиотек, созданных сторонними разработчиками, запуска анимаций, запроса данных или выполнения любых дополнительных настроек, которые потребуются перед отображением компонента на экране. Из этого метода можно вызвать метод `setState`, чтобы изменить состояние компонента непосредственно перед тем, как он будет первоначально отображен на экране.

Воспользуемся методом `componentWillMount` для инициализации запроса данных по ряду сотрудников. В случае удачного получения ответа мы обновим состояние. Помните промис `getFakeMembers`, который создали в главе 2? Применим его, чтобы загрузить произвольный список сотрудников с ресурса `randomuser.me`:

```
const getFakeMembers = count => new Promise((resolves, rejects) => {
  const api = `https://api.randomuser.me/?nat=US&results=${count}`
  const request = new XMLHttpRequest()
  request.open('GET', api)
  request.onload = () => (request.status == 200) ?
    resolves(JSON.parse(request.response).results) :
    reject(Error(request.statusText))
  request.onerror = err => rejects(err)
  request.send()
})
```

Данный промис мы используем в методе `componentWillMount`, принадлежащем компоненту списка сотрудников `MemberList`. Этот компонент задействует компонент `Member`, чтобы отобразить фотографию, имя, электронный адрес и место проживания каждого сотрудника:

```
const Member = ({ email, picture, name, location }) =>
  <div className="member">
    <img src={picture.thumbnail} alt="" />
    <h1>{name.first} {name.last}</h1>
    <p><a href="mailto:" + email>{email}</a></p>
    <p>{location.city}, {location.state}</p>
  </div>

class MemberList extends Component {
  constructor() {
    super()
    this.state = {
      members: [],
      loading: false,
      error: null
    }
  }

  componentWillMount() {
    this.setState({loading: true})
    getFakeMembers(this.props.count).then(
      members => {
        this.setState({members, loading: false})
      }
    )
  }
}
```

```
    },
    error => {
      this.setState({error, loading: false})
    }
  )
}

componentWillUpdate() {
  console.log('updating lifecycle')
}

render() {
  const { members, loading, error } = this.state
  return (
    <div className="member-list">
      {(loading) ?
        <span>Loading Members</span> :
        (members.length) ?
          members.map((user, i) =>
            <Member key={i} {...user} />
          ) :
          <span>0 members loaded...</span>
        }
      {(error) ? <p>Error Loading Members: error</p> : ""}
    </div>
  )
}
```

Изначально при установке компонента у `MemberList` будет пустой массив для сотрудников по имени `members`, а состояние загрузки `loading` будет иметь значение `false`. В методе `componentWillMount` состояние изменяется, чтобы отобразить факт выдачи запроса для загрузки данных о ряде сотрудников. Затем, во время ожидания завершения запроса, компонент отображается на экране. Поскольку теперь состояние `loading` имеет значение `true`, будет выведено сообщение, предупреждающее пользователя о задержке. Когда промис будет выполнен или даст сбой, состоянию `loading` будет возвращено значение `false` и либо загрузятся данные о сотрудниках, либо возвратится сообщение об ошибке. Вызов `setState` в этот момент приведет к повторному отображению нашего пользовательского интерфейса с данными о сотрудниках или с сообщением об ошибке.



Использование `setState` в `componentWillMount`

Вызов метода `setState` перед отображением компонента на экране не запустит жизненный цикл обновления, в отличие от вызова после отображения. Если этот метод вызывается внутри асинхронной функции обратного вызова, определенной в методе `componentWillMount`, то он будет вызван после отображения компонента на экране и запустит жизненный цикл обновления.

К другим методам жизненного цикла установки компонента относятся `componentDidMount` и `componentWillUnmount`. Первый вызывается сразу же после отображения компонента на экране, а второй — непосредственно перед удалением его с экрана.

Метод `componentDidMount` является еще одним местом, откуда удобно отправлять запросы к API. Он вызывается после отображения компонента на экране, поэтому любые вызовы `setState` из данного метода приведут к запуску жизненного цикла обновления и к повторному отображению компонента на экране.

Метод `componentDidMount` также хорошо подходит для инициализации любого кода JavaScript, созданного сторонними разработчиками, которому требуется DOM. Например, может потребоваться внедрить библиотеку перетаскивания объектов или библиотеку, обрабатывающую события прикосновения к экрану. Обычно таким библиотекам перед их инициализацией требуется DOM.

Кроме того, данный метод будет полезен при запуске фоновых процессов наподобие измерения интервалов времени или таймеров. Любые процессы, запущенные в `componentDidMount` или `componentWillMount`, могут быть прекращены в `componentWillUnmount`. От неиспользуемых фоновых процессов нужно своевременно избавляться.

Компоненты убираются с экрана, когда удаляются своими родительскими компонентами или с помощью метода `unmountComponentAtNode` из пакета `react-dom`. Этот метод используется для удаления корневого компонента. Когда удаляется корневой компонент, сначала удаляются его дочерние компоненты.

Рассмотрим пример с таймером. Он запускается при установке компонента `Clock`. Когда пользователь нажимает кнопку закрытия, таймер удаляется, для чего используется метод `unmountComponentAtNode`, и отсчет времени прекращается:

```
import React from 'react'
import { render, unmountComponentAtNode } from 'react-dom'
import { getClockTime } from './lib'
const { Component } = React
const target = document.getElementById('react-container')

class Clock extends Component {
  constructor() {
    super()
    this.state = getClockTime()
  }

  componentDidMount() {
    console.log("Starting Clock")
    this.ticking = setInterval(() =>
      this.setState(getClockTime())
    , 1000)
  }
}
```

```
componentWillUnmount() {
  clearInterval(this.ticking)
  console.log("Stopping Clock")
}

render() {
  const { hours, minutes, seconds, timeOfDay } = this.state
  return (
    <div className="clock">
      <span>{hours}</span>
      <span>:</span>
      <span>{minutes}</span>
      <span>:</span>
      <span>{seconds}</span>
      <span>{timeOfDay}</span>
      <button onClick={this.props.onClose}>x</button>
    </div>
  )
}

render(
  <Clock onClose={() => unmountComponentAtNode(target) }/>,
  target
)
```

В главе 3 мы создавали функцию `serializeTime`, которая абстрагирует из объекта данных показания времени в гражданском формате с лидирующими нулями. При каждом вызове `serializeTime` текущее время возвращается в объект, содержащий часы, минуты, секунды, а также индикатор a. m. или p. m. Изначально `serializeTime` вызывается для получения исходного состояния для наших часов.

После установки компонента запускается интервал, называемый `ticking`. Каждую секунду он запускает `setState` с новым показанием времени. Пользовательский интерфейс часов меняет свое значение для ежесекундного обновления времени.

При нажатии кнопки закрытия компонент `Clock` убирается с экрана. Непосредственно перед удалением часов из DOM интервал `ticking` сбрасывается, чтобы больше не запускаться в фоновом режиме.

Жизненный цикл обновления

Жизненный цикл обновления представляет собой последовательность методов, вызываемых при изменении состояния компонента или при получении от родительского компонента новых свойств. Этот жизненный цикл может использоваться для внедрения кода JavaScript перед обновлением компонента либо для взаимодействия с DOM после обновления. Кроме того, он может служить для повышения производительности приложения, поскольку дает возможность отменить ненужные обновления.

Жизненный цикл обновления запускается при каждом вызове `setState`. Вызов `setState` внутри жизненного цикла обновления станет причиной бесконечного рекурсивного цикла, который приведет к ошибке переполнения стека. Поэтому метод `setState` может вызываться только в методе `componentWillReceiveProps`, позволяющем компоненту обновлять состояние при обновлении его свойств.

В число методов жизненного цикла обновления входят:

- ❑ `componentWillReceiveProps(nextProps)` — вызывается только в случае передачи компоненту новых свойств; единственный метод, в котором может быть вызван метод `setState`;
- ❑ `shouldComponentUpdate(nextProps, nextState)` — привратник жизненного цикла обновления: предикат, способный отменить обновление; может использоваться для повышения производительности, разрешая только необходимые обновления;
- ❑ `componentWillUpdate(nextProps, nextState)` — вызывается непосредственно перед обновлением компонента; похож на метод `componentWillMount`, но вызывается только перед выполнением каждого обновления;
- ❑ `componentDidUpdate(prevProps, prevState)` — вызывается сразу же после выполнения обновления, после вызова метода отображения `render`; похож на метод `componentDidMount`, но вызывается только после каждого обновления.

Переделаем приложение — органайзер цветов, созданное в предыдущей главе. В частности, добавим к компоненту `Color` некие функции жизненного цикла обновления, которые позволят понаблюдать за работой этого цикла. Предположим, что у нас в массиве состояния уже есть четыре цвета: `ocean blue`, `tomato`, `lawn` и `party pink`. Сначала воспользуемся методом `componentWillMount` для инициализации объектов цвета с помощью стиля и установим для всех четырех элементов `Color` серый цвет фона:

```
import { Star, StarRating } from '../components'

export class Color extends Component {
  componentWillMount() {
    this.style = { backgroundColor: "#CCC" }
  }

  render() {
    const { title, rating, color, onRate } = this.props
    return
      <section className="color" style={this.style}>
        <h1 ref="title">{title}</h1>
        <div className="color"
          style={{ backgroundColor: color }}>
          </div>
          <StarRating starsSelected={rating}
            onRate={onRate} />
        </section>
  }
}
```

```
Color.propTypes = {
  title: PropTypes.string,
  rating: PropTypes.number,
  color: PropTypes.string,
  onRate: PropTypes.func
}

Color.defaultProps = {
  title: undefined,
  rating: 0,
  color: "#000000",
  onRate: f=>f
}
```

При начальной установке списка цветов все фоновые цвета будут серыми (рис. 7.1).



Рис. 7.1. Установленные цвета с серым фоном

К компоненту `Color` можно добавить метод `componentWillUpdate`, чтобы удалить серый фон из каждого цвета непосредственно перед обновлением последнего:

```
componentWillMount() {
  this.style = { backgroundColor: "#CCC" }
}

componentWillUpdate() {
```



```

    this.style = null
}

```

Добавление этих функций жизненного цикла обновления позволяет увидеть, когда компонент устанавливается и обновляется. Изначально установленные компоненты будут иметь серый фон. Как только произойдет обновление каждого цвета, цвет фона вернется к белому.

Если запустить этот код и оценить любой цвет, можно заметить обновление всех четырех цветов, пусть и была изменена оценка только одного из них (рис. 7.2).



Рис. 7.2. Оценка первого цвета запускает обновление, и происходит обновление всех четырех цветов

Здесь при смене оценки Ocean Blue с трех звезд на четыре обновляются все четыре цвета, потому что, когда родительский компонент `ColorList` обновляет состояние, он заново отображает каждый компонент `Color`. Заново отображаемые компоненты не подвергаются переустановке; если они уже присутствуют, то вместо этого происходит обновление. При обновлении компонента обновляются и все его дочерние компоненты. При оценке одного цвета обновляются все четыре цвета, все четыре компонента `StarRating` и все пять звезд в каждом компоненте.

Производительность нашего приложения можно повысить, предотвращая обновление цветов, когда их значения свойств не изменялись. Это можно сделать путем добавления в жизненный цикл обновления метода `shouldComponentUpdate`. Он возвращает либо `true`, либо `false` (`true` возвращается, когда компонент нужно обновить, а `false` — когда обновление должно быть пропущено):

```
componentWillMount() {
  this.style = { backgroundColor: "#CCC" }
}

shouldComponentUpdate(nextProps) {
  const { rating } = this.props
  return rating !== nextProps.rating
}

componentWillUpdate() {
  this.style = null
}
```

Метод `shouldComponentUpdate` может сравнить новые свойства со старыми. Новые передаются этому методу в виде аргумента, а старые по-прежнему берутся из текущих свойств, ведь компонент еще не обновлен. Если новый рейтинг остался таким же, как и в текущих свойствах, то цвет в обновлении не нуждается. При отсутствии обновления цвета не будет обновляться и ни один его дочерний компонент. Когда рейтинг не изменяется, не будет обновляться и все дерево компонентов под каждым компонентом `Color`.

Описанное выше можно продемонстрировать, запустив данный код и обновив любой из цветов. Метод `componentWillUpdate` будет вызван, только если компонент будет обновляться. В жизненном цикле он следует за методом `shouldComponentUpdate`. Фон останется серым, пока компоненты `Color` не будут обновлены путем изменения их рейтингов (рис. 7.3).

Если метод `shouldComponentUpdate` возвращает `true`, то далее следуют остальные действия жизненного цикла обновления. Остальные методы жизненного цикла обновления также получают в виде аргументов новые свойства и новое состояние. (Метод `componentDidUpdate` получает предыдущие свойства и предыдущее состояние, поскольку, если дело дошло до его применения, значит, обновление уже состоялось и свойства были изменены.)

Выведем в консоль сообщение после обновления компонента. В методе `componentDidUpdate` сравним текущие свойства со старыми, чтобы увидеть, повысился рейтинг или понизился:

```
componentWillMount() {
  this.style = { backgroundColor: "#CCC" }
}

shouldComponentUpdate(nextProps) {
```

```

    const { rating } = this.props
    return rating !== nextProps.rating
  }

  componentWillUpdate() {
    this.style = null
  }

  componentDidUpdate(prevProps) {
    const { title, rating } = this.props
    const status = (rating > prevProps.rating) ? 'better' : 'worse'
    console.log(`${title} is getting ${status}`)
  }
}

```



Рис. 7.3. С помощью метода `shouldComponentUpdate` происходит только одно обновление

Методы жизненного цикла обновления `componentWillUpdate` и `componentDidUpdate` великолепно подходят для взаимодействия с элементами DOM до и после обновлений. В следующем фрагменте кода процесс обновления будет приостановлен с выдачей оповещения, создаваемого в методе `componentWillUpdate`:

```

componentWillMount() {
  this.style = { backgroundColor: "#CCC" }
}

```

```

shouldComponentUpdate(nextProps) {
  return this.props.rating !== nextProps.rating
}

componentWillUpdate(nextProps) {
  const { title, rating } = this.props
  this.style = null
  this.refs.title.style.backgroundColor = "red"
  this.refs.title.style.color = "white"
  alert(`${title}: rating ${rating} -> ${nextProps.rating}`)
}

componentDidUpdate(prevProps) {
  const { title, rating } = this.props
  const status = (rating > prevProps.rating) ? 'better' : 'worse'
  this.refs.title.style.backgroundColor = ""
  this.refs.title.style.color = "black"
}

```

Если изменить рейтинг tomato с двух до четырех звезд, то процесс обновления приостановится с выдачей оповещения (рис. 7.4). Текущий элемент DOM для названия цвета получит другой фон и цвет текста.

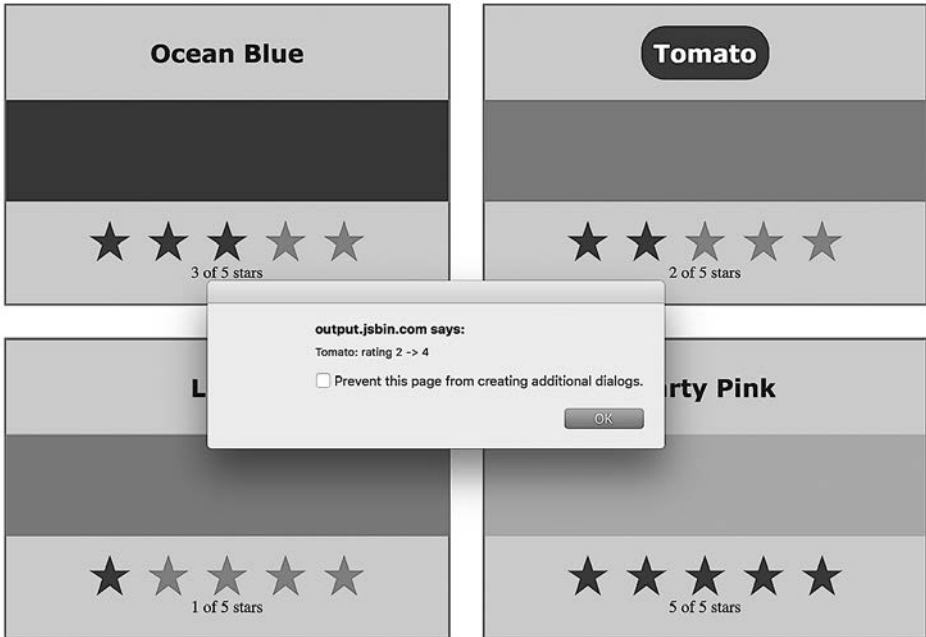


Рис. 7.4. Обновление приостановлено с выдачей оповещения

Как только оповещение будет убрано с экрана, компонент обновится и будет вызван метод `componentDidUpdate`, убирающий фоновый цвет названия (рис. 7.5).



Рис. 7.5. Метод `componentDidUpdate` удаляет выделение названия

Иногда наши компоненты сохраняют состояние, которое изначально устанавливается на основе свойств. Исходное состояние наших классов компонентов можно установить в конструкторе или в методе жизненного цикла `componentWillMount`. Когда эти свойства изменяются, приходится обновлять состояние с помощью метода `componentWillReceiveProps`.

В примере 7.1 имеется родительский компонент `HiddenMessages`, хранящий состояние скрытых сообщений. В состоянии этого компонента хранятся три сообщения, и всякий раз показывается только одно из них. Когда `HiddenMessages` устанавливается, к циклу прохождения сообщений добавляется интервал, позволяющий выводить эти сообщения только по одному.

Пример 7.1. Компонент `HiddenMessages`

```
class HiddenMessages extends Component {
  constructor(props) {
    super(props)
    this.state = {
      messages: [
        "The crow crows after midnight",
        "Bring a watch and dark clothes to the spot",
        "Jericho Jericho Go"
      ],
    },
```

```

        showing: -1
    }
}

componentWillMount() {
    this.interval = setInterval(() => {
        let { showing, messages } = this.state
        showing = (++showing >= messages.length) ?
            -1 :
            showing
        this.setState({showing})
    }, 1000)
}

componentWillUnmount() {
    clearInterval(this.interval)
}

render() {
    const { messages, showing } = this.state
    return (
        <div className="hidden-messages">
            {messages.map((message, i) =>
                <HiddenMessage key={i}
                    hide={{!==(i===showing)}}
                    {message}
                </HiddenMessage>
            )}
        </div>
    )
}
}

```

Компонент `HiddenMessages` совершает циклический обход каждого сообщения в массиве состояния. Соответствующая логика определяется в методе `componentWillMount`. Когда компонент устанавливается, добавляется интервал, через который обновляется индекс отображаемого сообщения. Компонент отображает на экране все сообщения, используя компонент `HiddenMessage`, и устанавливает свойства `hide` одного из них в `true` в каждом цикле. Для остальных свойств устанавливается значение `false` и скрытые сообщения меняются каждую секунду.

Посмотрите на компонент `HiddenMessage`, который применяется для каждого сообщения (пример 7.2). При начальной установке этого компонента для определения его состояния используется свойство `hide`. Но когда родительский компонент обновляет эти свойства компонентов, ничего не происходит. Данный компонент ничего об этом не знает.

Пример 7.2. Компонент `HiddenMessage`

```

class HiddenMessage extends Component {
    constructor(props) {

```

```

    super(props)
    this.state = {
      hidden: (props.hide) ? props.hide : true
    }
  }

  render() {
    const { children } = this.props
    const { hidden } = this.state
    return (
      <p>
        {(hidden) ?
          children.replace(/[a-zA-Z0-9]/g, "x") :
          children
        }
      </p>
    )
  }
}

```

Проблема возникает, когда родительский компонент изменяет значение свойства `hide`. Это изменение не вызывает автоматического изменения состояния `HiddenMessage`.

Для подобных сценариев был создан метод жизненного цикла `componentWillReceiveProps`. Он будет вызываться, когда родительский компонент изменил свойства, и они могут использоваться для изменения состояния изнутри:

```

class HiddenMessage extends Component {
  constructor(props) {
    super(props)
    this.state = {
      hidden: (props.hide) ? props.hide : true
    }
  }

  componentWillReceiveProps(nextProps) {
    this.setState({hidden: nextProps.hide})
  }

  render() {
    const { children } = this.props
    const { hidden } = this.state
    return (
      <p>
        {(hidden) ?
          children.replace(/[a-zA-Z0-9]/g, "x") :
          children
        }
      </p>
    )
  }
}

```

Когда родительский компонент `HiddenMessages` изменяет значение свойства `hide`, метод `componentWillReceiveProps` позволяет обновить состояние.

Установка состояния из свойств

Предыдущий фрагмент кода был сокращен до демонстрации применения `componentWillReceiveProps`. Если больше с компонентом `HiddenMessage` ничего не нужно делать, то вместо этого следует воспользоваться функциональным компонентом, не имеющим состояния. Единственная причина, по которой когда-либо придется добавлять состояние к дочернему компоненту, может быть связана с необходимостью что-либо изменять внутри данного компонента.

Например, использование метода `componentWillReceiveProps` для изменения состояния было бы оправданно в случае, если бы компоненту требовался вызов `setState`:

```
hide() {
  const hidden = true
  this.setState({hidden})
}

show() {
  const hidden = false
  this.setState({hidden})
}

return
  <p onMouseEnter={this.show}
    onMouseLeave={this.hide}>
    {(hidden) ?
      children.replace(/[a-zA-Z0-9]/g, "x") :
      children
    }
  </p>
```

В таком случае было бы целесообразно сохранять состояние в компоненте `HiddenMessage`. Если компонент не собирается изменять самого себя, то его нужно оградить от состояния и управлять последним только из родительского компонента.

Методы жизненного цикла компонента позволяют расширить контроль над тем, как компонент должен быть отображен на экране или обновлен. Они предоставляют средство, позволяющее добавлять функциональные возможности до и после того, как произойдет установка или обновление. Далее мы обсудим, как применять методы жизненного цикла для внедрения библиотек JavaScript, созданных сторонними разработчиками. Но сначала вкратце рассмотрим API `React.Children`.

React.Children

API `React.Children` предоставляет способ работы с дочерними компонентами отдельно взятого компонента. Он позволяет вести их подсчет, выполнять их отображение, циклический обход или превращать `props.children` в массив. Он также позволяет с помощью `React.Children.only` убедиться, что на экран выводится единственный дочерний компонент:


```

import { Children, PropTypes } from 'react'
import { render } from 'react-dom'

const Display = ({ ifTruthy=true, children }) =>
  (ifTruthy) ?
    Children.only(children) :
    null

const age = 22

render(
  <Display ifTruthy={age >= 21}>
    <h1>You can enter</h1>
  </Display>,
  document.getElementById('react-container')
)

```

В этом примере компонент `Display` будет показывать только один дочерний компонент, элемент `h1`. Если компонент `Display` содержит несколько дочерних компонентов, то React выдаст сообщение об ошибке: `onlyChild must be passed a children with exactly one child` (`onlyChild` должны передаваться дочерние компоненты, имеющие только один дочерний компонент).

API `React.Children` может использоваться и для преобразования свойства `children` в массив. В следующем фрагменте кода компонент `Display` расширяется, чтобы дополнительно обрабатывать другие случаи:

```

const { Children, PropTypes } = React
const { render } = ReactDOM

const findChild = (children, child) =>
  Children.toArray(children)
    .filter(c => c.type === child)[0]

const WhenTruthy = ({children}) =>
  Children.only(children)

const WhenFalsy = ({children}) =>
  Children.only(children)

const Display = ({ ifTruthy=true, children }) =>
  (ifTruthy) ?
    findChild(children, WhenTruthy) :
    findChild(children, WhenFalsy)

const age = 19

render(
  <Display ifTruthy={age >= 21}>
    <WhenTruthy>
      <h1>You can Enter</h1>
    </WhenTruthy>
    <WhenFalsy>
      <h1>Beat it Kid</h1>

```

```

    </WhenFalsy>
  </Display>,
  document.getElementById('react-container')
)

```

Компонент `Display` выведет на экран один дочерний элемент, когда условие вычисляется в `true`, или другой, когда вычисляется в `false`. Для этого создаются компоненты `WhenTruthy` и `WhenFalsy`, которые применяются в компоненте `Display` в качестве дочерних. Функция `findChild` использует `React.Children` для преобразования `children` в массив. Последний можно отфильтровать, чтобы найти и вернуть отдельно взятый дочерний компонент по типу компонента.

Подключение библиотек JavaScript

Такие фреймворки, как `Angular` и `jQuery`, поставляются со своими собственными средствами доступа к данным, отображения пользовательского интерфейса, моделирования состояния и т. д. В отличие от них `React` является простой библиотекой для создания представлений, поэтому может потребоваться взаимодействие с другими библиотеками JavaScript. Если хорошо разбираться в работе функций жизненного цикла, то можно заставить `React` поладить практически с любой библиотекой JavaScript.



React с jQuery

Вообще-то использование `jQuery` с `React` не находит одобрения со стороны сообщества. И все же объединить `jQuery` и `React` вполне возможно, и это может быть неплохим вариантом для изучения `React` или перевода устаревшего кода на `React`. Но приложения работают намного лучше, если к `React` подключаются библиотеки, имеющие меньший объем, чем большие рабочие среды. Кроме того, при использовании `jQuery` для непосредственного воздействия на `DOM` игнорируется применение виртуальной `DOM`, что может привести к возникновению непонятных ошибок.

В этом разделе к компонентам `React` мы подключим несколько различных библиотек JavaScript. А именно, рассмотрим способы выполнения вызовов функций API и визуализации данных с поддержкой других библиотек JavaScript.

Создание запросов с помощью Fetch

`Fetch` представляет собой полифилл, созданный группой WHATWG и упрощающий выполнение вызовов API с использованием промисов. В этом подразделе будет представлена `isomorphic-fetch`, версия `Fetch`, которая хорошо работает с `React`. Установим ее:

```
npm install isomorphic-fetch --save
```

Функции жизненного цикла компонента предоставляют место для подключения JavaScript. В данном случае мы будем использовать это место для выполнения вызовов API. Компоненты, выполняющие данные вызовы, должны справляться с *задержками*, с которыми сталкивается пользователь, ожидая ответа. Эти проблемы можно переложить на наше состояние, включив переменные, сообщающие компоненту, находится ли запрос в режиме ожидания.

В следующем примере компонентом `CountryList` создается упорядоченный список названия стран. После установки компонент выполняет вызов API и изменяет состояние, чтобы отобразить загрузку данных. Состояние `loading` имеет значение `true` до тех пор, пока не поступит ответ на этот вызов:

```
import { Component } from 'react'
import { render } from 'react-dom'
import fetch from 'isomorphic-fetch'

class CountryList extends Component {
  constructor(props) {
    super(props)
    this.state = {
      countryNames: [],
      loading: false
    }
  }

  componentDidMount() {
    this.setState({loading: true})
    fetch('https://restcountries.eu/rest/v1/all')
      .then(response => response.json())
      .then(json => json.map(country => country.name))
      .then(countryNames =>
        this.setState({countryNames, loading: false})
      )
  }

  render() {
    const { countryNames, loading } = this.state
    return (loading) ?
      <div>Loading Country Names...</div> :
      (!countryNames.length) ?
        <div>No country Names</div> :
        <ul>
          {countryNames.map(
            (x,i) => <li key={i}>{x}</li>
          )}
        </ul>
    }
  }
}

render(
  <CountryList />,
  document.getElementById('react-container')
)
```

Когда компонент устанавливается, непосредственно перед вызовом `fetch` состоянию `loading` присваивается значение `true`. Тем самым мы оповещаем компонент, а в конечном итоге и пользователей, что находимся в процессе извлечения данных. Когда приходит ответ от вызова `fetch`, мы получаем JSON-объект, который отображаем на массив с названиями стран. И наконец, названия стран добавляются к состоянию и DOM обновляется.

Подключение D3 Timeline

Data Driven Documents (D3) является JavaScript-фреймворком, которым можно воспользоваться, чтобы создать визуализацию данных для браузера. D3 предоставляет широкий набор инструментов, позволяющих масштабировать и интерполировать данные. Кроме того, D3 имеет выраженный функциональный характер. D3-приложения состояются из цепочки вызовов функций с целью создания визуализации DOM из массива данных.

Одним из примеров визуализации данных является шкала времени (*timeline*). Она получает в качестве данных даты событий и создает визуальное представление этой информации с помощью элементов графики. Исторические события, которые были раньше, представлены левее событий, имевших место позже. Пространство между каждым событием на шкале времени, заданное в пикселах, представляет собой время, прошедшее между событиями (рис. 7.6).

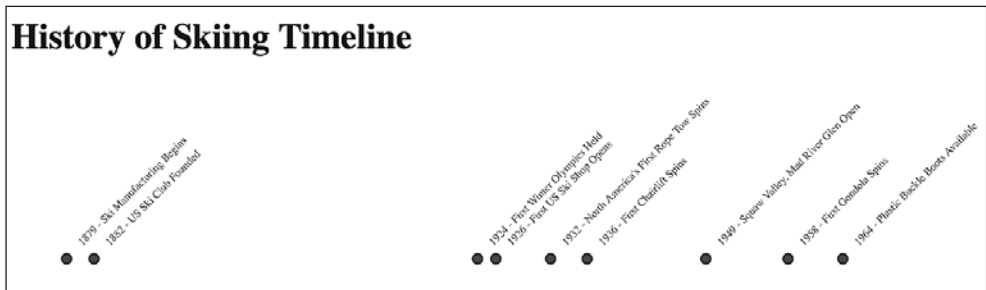


Рис. 7.6. Визуализация данных на шкале времени

Эта шкала отражает визуализацию почти столетней истории всего лишь в пятистах пикселах. Процесс преобразования прошедших лет в соответствующие им пиксельные значения называется *интерполяцией*. D3 предоставляет все инструменты для интерполяции диапазонов данных от одних единиц измерения к другим.

Посмотрим на то, как подключить D3 к React, чтобы создать эту шкалу времени. Сначала нужно установить D3:

```
npm install d3 --save
```

Библиотека D3 получает данные, обычно в виде массивов объектов, и на их основе выработывает визуализацию. Посмотрите на массив исторических данных, касающихся лыж. Именно он предоставляет данные для нашей шкалы времени:

```
const historicDatesForSkiing = [
  {
    year: 1879,
    event: "Ski Manufacturing Begins"
  },
  {
    year: 1882,
    event: "US Ski Club Founded"
  },
  {
    year: 1924,
    event: "First Winter Olympics Held"
  },
  {
    year: 1926,
    event: "First US Ski Shop Opens"
  },
  {
    year: 1932,
    event: "North America's First Rope Tow Spins"
  },
  {
    year: 1936,
    event: "First Chairlift Spins"
  },
  {
    year: 1949,
    event: "Squaw Valley, Mad River Glen Open"
  },
  {
    year: 1958,
    event: "First Gondola Spins"
  },
  {
    year: 1964,
    event: "Plastic Buckle Boots Available"
  }
]
```

Проще всего подключить D3 к компоненту React, позволив библиотеке отобразить пользовательский интерфейс, а затем заставив D3 создать и добавить визуализацию. В следующем примере D3 подключается к компоненту React. После отображения его на экране D3 выстраивает визуализацию и добавляет ее к DOM:

```
import d3 from 'd3'
import { Component } from 'react'
import { render } from 'react-dom'

class Timeline extends Component {
  constructor({data=[]}) {
```

```

    const times = d3.extent(data.map(d => d.year))
    const range = [50, 450]
    super({data})
    this.state = {data, times, range}
  }

  componentDidMount() {
    let group
    const { data, times, range } = this.state
    const { target } = this.refs
    const scale = d3.time.scale().domain(times).range(range)

    d3.select(target)
      .append('svg')
      .attr('height', 200)
      .attr('width', 500)

    group = d3.select(target.children[0])
      .selectAll('g')
      .data(data)
      .enter()
      .append('g')
      .attr(
        'transform',
        (d, i) => `translate(' + scale(d.year) + ', 0)`
      )

    group.append('circle')
      .attr('cy', 160)
      .attr('r', 5)
      .style('fill', 'blue')

    group.append('text')
      .text(d => d.year + " - " + d.event)
      .style('font-size', 10)
      .attr('y', 115)
      .attr('x', -95)
      .attr('transform', 'rotate(-45)')
  }

  render() {
    return (
      <div className="timeline">
        <h1>{this.props.name} Timeline</h1>
        <div ref="target"></div>
      </div>
    )
  }
}

render(
  <Timeline name="History of Skiing"
    data={historicDatesForSkiing} />,
  document.getElementById('react-container')
)

```

В этом примере часть настроек D3 устанавливается в конструкторе, но все самое основное делается средствами D3 в функции `componentDidMount`. Как только DOM отобразится на экране, D3 выстраивает визуализацию, используя масштабируемую векторную графику (*scalable vector graphics*, SVG). Это вполне работоспособный подход и весьма неплохой способ быстрого подключения D3-визуализации к компонентам React.

Но эту связку можно углубить, позволив React управлять математическими расчетами в DOM и D3. Посмотрим на следующие три строки кода:

```
const times = d3.extent(data.map(d => d.year))
const range = [50, 450]

const scale = d3.time.scale().domain(times).range(range)
```

И `times`, и `range` получили значения в конструкторе и были добавлены к состоянию компонента. Нашу информационную область представляет `times`, где содержатся значения для самого раннего и самого последнего годов. Вычисление выполняется с помощью D3-функции `extent`, чтобы найти минимальное и максимальное значение в массиве числовых значений. Диапазон для шкалы времени в пикселах представляет `range`. Первая дата, 1879, будет помещена на шкале x на отметке 50 px, а последняя, 1964, — на шкале x на отметке 450 px.

В последней строке создается `scale`, представляющая собой функцию, которой можно воспользоваться для интерполяции пиксельного значения для любого года на шкале времени. Функция `scale` создается путем отправки `domain` и `range` D3-функции `time.scale`. Функция `scale` используется в визуализации для получения позиции x для каждой даты, попадающей в промежуток от 1879 до 1964 года.

Вместо создания `scale` в `componentDidMount` эту функцию можно добавить к компоненту в конструкторе после появления `domain` и `range`. Теперь доступ к `scale` может быть получен из любого места компонента с помощью выражения `this.scale(year)`:

```
constructor({data=[]}) {
  const times = d3.extent(data.map(d => d.year))
  const range = [50, 450]
  super({data})
  this.scale = d3.time.scale().domain(times).range(range)
  this.state = {data, times, range}
}
```

Внутри `componentDidMount` средствами D3 сначала создается SVG-элемент, который затем добавляется к целевой ссылке:

```
d3.select(target)
  .append('svg')
  .attr('height', 200)
  .attr('width', 500)
```

Построение пользовательского интерфейса возлагается на React. Вместо использования для этой цели D3 создадим компонент холста `Canvas`, возвращающий SVG-элемент:

```
const Canvas = ({children}) =>
  <svg height="200" width="500">
    {children}
  </svg>
```

Далее D3 выбирает элемент `svg`, являющийся первым дочерним элементом ниже цели, и добавляет элемент `group` для каждой точки данных нашего массива шкалы времени. Затем этот элемент позиционируется путем преобразования значения по оси *X* с помощью функции `scale`:

```
group = d3.select(target.children[0])
  .selectAll('g')
  .data(data)
  .enter()
  .append('g')
  .attr(
    'transform',
    (d, i) => `translate(${scale(d.year)} + ${0})`
  )
```

Элемент `group` является элементом DOM, следовательно, можно позволить React справляться и с этой задачей. Рассмотрим компонент `TimelineDot`, пригодный для настройки элементов `group` и позиционирования их на оси *X*:

```
const TimelineDot = ({position}) =>
  <g transform={`translate(${position},0)`}></g>
```

Затем D3 добавляет к `group` элемент кружочка `circle` и некоторый «стиль». Элемент `text` получает свое значение путем объединения года события с описанием этого события. Затем средствами библиотеки выполняются позиционирование и поворот данного текста относительно синего кружочка:

```
group.append('circle')
  .attr('cy', 160)
  .attr('r', 5)
  .style('fill', 'blue')

group.append('text')
  .text(d => d.year + " - " + d.event)
  .style('font-size', 10)
  .attr('y', 115)
  .attr('x', -95)
  .attr('transform', 'rotate(-45)')
```

Нам остается лишь внести изменения в компонент `TimelineDot`, включив в него элемент `circle` и элемент `text`, который извлекает текст из свойств:

```
const TimelineDot = ({position, txt}) =>
  <g transform={`translate(${position},0)`}>
```



```

<circle cy={160}
        r={5}
        style={{fill: 'blue'}} />
<text y={115}
      x={-95}
      transform="rotate(-45)"
      style={{fontSize: '10px'}}>{txt}</text>

</g>

```

Теперь React отвечает за управление пользовательским интерфейсом с помощью виртуальной DOM. Роль библиотеки D3 была урезана, но она по-прежнему предоставляет ряд важных функциональных возможностей, не имеющих в React. D3 помогает создать информационную область и диапазон (`domain` и `range`) и вводит функцию `scale`, которой можно воспользоваться для интерполяции пиксельных значений из прошедших лет.

Наш реструктурированный компонент `Timeline` может приобрести следующий вид:

```

class Timeline extends Component {
  constructor({data=[]}) {
    const times = d3.extent(data.map(d => d.year))
    const range = [50, 450]
    super({data})
    this.scale = d3.time.scale().domain(times).range(range)
    this.state = {data, times, range}
  }

  render() {
    const { data } = this.state
    const { scale } = this
    return (
      <div className="timeline">
        <h1>{this.props.name} Timeline</h1>
        <Canvas>
          {data.map((d, i) =>
            <TimelineDot position={scale(d.year)}
                        txt={` ${d.year} - ${d.event}` }
            />
          )}
        </Canvas>
      </div>
    )
  }
}

```

Подключить к React можно практически любую библиотеку JavaScript. Местом, где React может уступить свои полномочия другому коду JavaScript, могут стать функции жизненного цикла. Но при этом нужно избегать добавления библиотек, управляющих пользовательским интерфейсом: эта работа React.

Компоненты высшего порядка

Компонент высшего порядка (higher-order component, НОС) представляет собой простую функцию, получающую в виде аргумента один компонент React и возвращающую другой. Обычно НОС заключают поступающий компонент в класс, обслуживают состояние или обладают функциональностью. Компоненты высшего порядка — наилучший способ многократного использования функциональных возможностей компонентов React.

Миксины больше не поддерживаются

До выхода React v0.13 наилучший способ встраивания функциональных возможностей в компонент React был связан с использованием *миксинов*. Они могли добавляться непосредственно к компонентам, создаваемым с помощью метода `createClass` в качестве свойства конфигурации. Миксины по-прежнему могут применяться с методом `React.createClass`, но не поддерживаются в классах ES6 или в функциональных компонентах, не имеющих состояния. Они также не будут поддерживаться будущими версиями React.

НОС позволяют заключать один компонент в другой. Родительский компонент может сохранять состояние или содержать функциональность, пригодную для передачи вниз составному компоненту в виде свойств. Этому составному компоненту ничего не нужно знать о реализации НОС имен свойств и методов, к которым он открывает доступ.

Рассмотрим компонент `PeopleList`. Он загружает данные о произвольных пользователях из API и отображает список имен. При загрузке данных пользователей выводится сообщение о загрузке. После нее эти данные отображаются в DOM:

```
import { Component } from 'react'
import { render } from 'react-dom'
import fetch from 'isomorphic-fetch'

class PeopleList extends Component {
  constructor(props) {
    super(props)
    this.state = {
      data: [],
      loaded: false,
      loading: false
    }
  }

  componentWillMount() {
    this.setState({loading:true})
    fetch('https://randomuser.me/api/?results=10')
      .then(response => response.json())
      .then(obj => obj.results)
  }
}
```

```

        .then(data => this.setState({
            loaded: true,
            loading: false,
            data
        })))
    }

    render() {
        const { data, loading, loaded } = this.state
        return (loading) ?
            <div>Loading...</div> :
            <ol className="people-list">
                {data.map((person, i) => {
                    const {first, last} = person.name
                    return <li key={i}>{first} {last}</li>
                })}
            </ol>
        }
    }
}

render(
    <PeopleList />,
    document.getElementById('react-container')
)

```

Для загрузки данных о пользователях из API JSON компонент `PeopleList` включает вызов функции `getJSON` из библиотеки `jQuery`. Когда компонент отображается на экране, он показывает сообщение о загрузке или выводит список имен, в зависимости от того, имеет свойство загрузки значение `true` или нет.

Если эту функциональность загрузки приспособить должным образом, то можно будет многократно использовать ее среди компонентов. Можно создать компонент высшего порядка по имени `DataComponent`, который может применяться для создания компонентов React, загружающих данные. Чтобы задействовать `DataComponent`, уберем из компонента `PeopleList` состояние и создадим функциональный компонент, не имеющий состояния, который получает данные через свойства:

```

import { render } from 'react-dom'

const PeopleList = ({data}) =>
    <ol className="people-list">
        {data.results.map((person, i) => {
            const {first, last} = person.name
            return <li key={i}>{first} {last}</li>
        })}
    </ol>

const RandomMeUsers = DataComponent(
    PeopleList,
    "https://randomuser.me/api/"
)

render(

```

```

    <RandomMeUsers count={10} />,
    document.getElementById('react-container')
  )

```

Теперь можно создать компонент `RandomMeUsers`, позволяющий загружать и выводить на экран имена пользователей из одного и того же источника, `randomuser.me`. Нам останется лишь предоставить количество пользователей, чьи данные нужно загрузить. Обработка данных была перенесена в НОС, а пользовательский интерфейс управляется компонентом `PeopleList`. НОС предоставляет состояние для загрузки и механизм для загрузки данных и изменения собственного состояния. Пока данные загружаются, этот компонент выводит сообщение о загрузке. По окончании загрузки данных НОС управляет установкой компонента `PeopleList` и передает ему данные о людях в виде свойства `data`:

```

const DataComponent = (ComposedComponent, url) =>
  class DataComponent extends Component {
    constructor(props) {
      super(props)
      this.state = {
        data: [],
        loading: false,
        loaded: false
      }
    }

    componentWillMount() {
      this.setState({loading:true})
      fetch(url)
        .then(response => response.json())
        .then(data => this.setState({
          loaded: true,
          loading: false,
          data
        })))
    }

    render() {
      return (
        <div className="data-component">
          {(this.state.loading) ?
            <div>Loading...</div> :
            <ComposedComponent {...this.state} />}
        </div>
      )
    }
  }

```

Обратите внимание: компонент `DataComponent` по сути является функцией. Все компоненты высшего порядка — функции. Компонентом, который мы будем заключать, будет `ComposedComponent`. Возвращенный класс, `DataComponent`, сохраняет состояние и управляет им. Когда это состояние изменяется и данные уже загружены на экран, отображается `ComposedComponent` и эти данные передаются ему в виде свойства.

Этот НОС пригоден для создания компонента данных любого типа. Посмотрим, как компонент `DataComponent` можно использовать повторно для добавления компонента `CountryDropDown`, заполняемого названиями разных стран, получаемыми из API `restcountries.eu`:

```
import { render } from 'react-dom'

const CountryNames = ({data, selected=""}) =>
  <select className="people-list" defaultValue={selected}>
    {data.map(({name}, i) =>
      <option key={i} value={name}>{name}</option>
    )}
  </select>

const CountryDropDown =
  DataComponent(
    CountryNames,
    "https://restcountries.eu/rest/v1/all"
  )

render(
  <CountryDropDown selected="United States" />,
  document.getElementById('react-container')
)
```

Компонент `CountryNames` получает названия стран через свойства. Компонент `DataComponent` управляет загрузкой и передачей информации о каждой стране.

Обратите внимание: у компонента `CountryNames` также имеется свойство `selected`. Оно заставит компонент выбрать по умолчанию `United States`. Но пока данное свойство не работает. Мы не передали свойства составному компоненту из нашего НОС.

Внесем изменения в НОС, чтобы передать любые получаемые им свойства ниже, в адрес составного компонента:

```
render() {
  return (
    <div className="data-component">
      {(this.state.loading) ?
        <div>Loading...</div> :
        <ComposedComponent {...this.state}
                           {...this.props} />
      }
    </div>
  )
}
```

Теперь НОС передает состояние и свойства вниз составному компоненту. Если мы запустим этот код, то увидим, что заранее выбранной страной в компоненте `CountryDropDown` будет `United States`.

Рассмотрим еще один НОС. Ранее в этой главе был разработан компонент скрытых сообщений `HiddenMessage`. Возможность показа или скрытия содержимого

относится к числу того, что может использоваться повторно. Следующий пример включает НОС `Expandable`, функционирующий аналогично компоненту `HiddenMessage`. Содержимое можно показывать или скрывать в зависимости от значения булева свойства `collapsed`. Этот НОС также предоставляет механизм для переключения свойства `collapsed` из одного состояния в другое (пример 7.3).

Пример 7.3. `./components/hoc/Expandable.js`

```
import { Component } from 'react'

const Expandable = ComposedComponent =>
  class Expandable extends Component {

    constructor(props) {
      super(props)
      const collapsed =
        (props.hidden && props.hidden === true) ?
          true :
          false
      this.state = {collapsed}
      this.expandCollapse = this.expandCollapse.bind(this)
    }

    expandCollapse() {
      let collapsed = !this.state.collapsed
      this.setState({collapsed})
    }

    render() {
      return <ComposedComponent
        expandCollapse={this.expandCollapse}
        {...this.state}
        {...this.props} />
    }
  }
```

НОС `Expandable` получает компонент `ComposedComponent` и заключает его в состояние и функциональные средства, позволяющие ему показывать или скрывать содержимое. Изначально состояние `collapsed` устанавливается с помощью поступающих свойств или значения по умолчанию `false`. Состояние `collapsed` передается ниже, в адрес компонента `ComposedComponent`, в виде свойства.

У этого компонента также имеется метод для переключения состояния, который называется `expandCollapse`. Данный метод тоже передается ниже, в адрес компонента `ComposedComponent`. После вызова он изменит состояние `collapsed` и обновит компонент `ComposedComponent` этим состоянием.

Если свойства `DataComponent` изменяются родительским компонентом, то данный компонент обновит состояние `collapsed` и передаст новое состояние вниз, компоненту `ComposedComponent`, в виде свойства.

Наконец, и состояние и свойства передаются вниз, компоненту `ComposedComponent`. Теперь НОС можно применять для создания нескольких новых компонентов. Сначала воспользуемся им для добавления компонента `HiddenMessage`, который был определен ранее в данной главе:

```
const ShowHideMessage = ({children, collapsed, expandCollapse}) =>
  <p onClick={expandCollapse}>
    {(collapsed) ?
      children.replace(/[a-zA-Z0-9]/g, "x") :
      children}
  </p>

const HiddenMessage = Expandable(ShowHideMessage)
```

Здесь создается компонент `HiddenMessage`, который будет заменять каждую букву или цифру в строке буквой `x`, если свойство `collapsed` имеет значение `true`. Когда значением свойства `collapsed` является `false`, будет показано сообщение. Испытайте данный компонент в ранее определенном в этой главе компоненте `HiddenMessages`.

Воспользуемся этим же НОС для создания кнопки, показывающей и свертывающей скрываемое содержимое, находящееся в контейнере `div`. В следующем примере компонент `MenuButton` можно использовать для создания компонента `PopUpButton`, переключающего отображение содержимого:

```
class MenuButton extends Component {
  componentWillReceiveProps(nextProps) {
    const collapsed =
      (nextProps.collapsed && nextProps.collapsed === true) ?
      true :
      false
    this.setState({collapsed})
  }

  render() {
    const {children, collapsed, txt, expandCollapse} = this.props
    return (
      <div className="pop-button">
        <button onClick={expandCollapse}>{txt}</button>
        {(!collapsed) ?
          <div className="pop-up">
            {children}
          </div> :
          ""
        }
      </div>
    )
  }
}

const PopUpButton = Expandable(MenuButton)

render(
```

```

<PopUpButton hidden={true} txt="toggle popup">
  <h1>Hidden Content</h1>
  <p>This content will start off hidden</p>
</PopUpButton>,
document.getElementById('react-container')
)

```

Компонент `PopUpButton` создается с помощью компонента `MenuButton`. Он будет передавать состояние свернутости, а также функцию для изменения состояния компоненту `MenuButton` в виде свойств. Когда пользователи нажимают кнопку, этот компонент будет вызывать функцию `expandCollapse` и переключать состояние свернутости. Когда состояние имеет значение свернутости, показывается только кнопка. В противном случае показываются кнопка и контейнер `div` с ранее скрытым содержимым.

Компоненты высшего порядка предоставляют весьма удобный способ повторного использования функциональных возможностей с абстрагированием от подробностей того, как происходит управление состоянием компонента или жизненным циклом этого компонента. НОС позволяют создавать больше функциональных компонентов, не имеющих состояния и ответственных лишь за пользовательский интерфейс.

Управление состоянием за пределами React

Управление состоянием в React нас вполне устраивает. С помощью встроенной в библиотеку системы управления состоянием можно создать множество разнообразных приложений. Но когда ваши приложения увеличиваются в объеме, следить за их состоянием становится сложнее. Хранение состояния в одном месте, а именно в корне вашего дерева компонентов, может упростить решение задачи, но даже при этом приложение может дорасти до такого объема, при котором логичнее было бы изолировать данные состояния на их собственном уровне, независимо от пользовательского интерфейса.

Одним из преимуществ управления состоянием за пределами React является сокращение или даже полный отказ от использования компонентов, создаваемых с помощью классов. Если состояние не применяется, то становится проще сохранять большинство за компонентами, не имеющими состояния. Создавать класс нужно будет только при востребованности функций жизненного цикла. И даже тогда функциональность, связанную с классом, можно изолировать в НОС, оставляя компоненты, содержащие только пользовательский интерфейс, не имеющими состояния. В функциональных компонентах, не имеющих состояния, проще разобратся, и их легче тестировать. Они относятся к чистым функциям, поэтому хорошо вписываются в понятие строго функциональных приложений.

Под управлением состоянием за пределами React может подразумеваться множество разнообразных вариантов. React можно использовать с моделями `Backbone` или с любой другой MVC-библиотекой, моделирующей состояние. Управлять

состоянием можно с помощью собственной системы, а также глобальных переменных или локального хранилища `localStorage` и обычного JavaScript. Управление состоянием за пределами React просто означает, что в ваших приложениях не меняется состояние React или метод `setState`.

Отображение часов. Ранее, в главе 3, на основе правил функционального программирования мы создали тикающие часы. Все приложение содержит функции и функции высшего порядка, входящие в состав более крупных функций, составляющих функцию `startTicking`, запускающую часы и отображающую время в консоли:

```
const startTicking = () =>
  setInterval(
    compose(
      clear,
      getCurrentTime,
      abstractClockTime,
      convertToCivilianTime,
      doubleDigits,
      formatClock("hh:mm:ss tt"),
      display(log)
    ),
    oneSecond()
  )

startTicking()
```

А что, если вместо отображения времени в консоли мы выведем его в окно браузера? Для вывода показаний часов в контейнере `div` можно создать следующий компонент React:

```
const AlarmClockDisplay = ({hours, minutes, seconds, ampm}) =>
  <div className="clock">
    <span>{hours}</span>
    <span>:</span>
    <span>{minutes}</span>
    <span>:</span>
    <span>{seconds}</span>
    <span>{ampm}</span>
  </div>
```

Этот компонент получает в свойствах значения часов, минут, секунд и времени суток. Затем создает DOM, где эти свойства могут быть отображены.

Метод `log` можно заменить методом `render` и отправить наш компонент, который будет использоваться для отображения времени в гражданском формате с лидирующими нулями, добавленными к значениям меньше 10:

```
const startTicking = () =>
  setInterval(
    compose(
      getCurrentTime,
```

```

        abstractClockTime,
        convertToCivilianTime,
        doubleDigits,
        render(AlarmClockDisplay)
    ),
    oneSecond()
)
startTicking()

```

Метод `render` должен быть функцией высшего порядка. Ему нужно будет изначально получить в виде свойства компонент `AlarmClockDisplay`, когда с ним компонуется и к нему прикрепляется метод `startTicking`. Со временем методу `render` нужно будет воспользоваться этим компонентом для ежесекундного отображения показаний отформатированного времени:

```

const render = Component => civilianTime =>
  ReactDOM.render(
    <Component {...civilianTime} />,
    document.getElementById('react-container')
  )

```

Функция высшего порядка для `render` ежесекундно вызывает `ReactDOM.render` и обновляет DOM. Такой подход позволяет задействовать преимущество присутствующего React ускоренного отображения DOM, но при этом не требует применения компонента, созданного с применением класса и использующего состояние.

Состояние приложения управляется за пределами React. Эта библиотека позволяет нам сохранить в неприкосновенности функциональную архитектуру за счет нашей собственной функции высшего порядка, отображающей компонент с помощью `ReactDOM.render`. Управлять состоянием за пределами React не обязательно, это всего лишь еще один вариант. React является библиотекой, и решение о том, как наилучшим образом воспользоваться ею в вашем приложении, зависит целиком от вас самих.

Далее мы приступим к рассмотрению модели конструирования Flux, созданной в качестве альтернативы управлению состоянием в React.

Flux

Flux — модель конструирования, разработанная в Facebook с целью организовать однонаправленный поток данных. До ее появления в архитектуре веб-разработки преобладали различные вариации модели конструирования MVC. Flux является альтернативой MVC, представляя собой абсолютно другую модель конструирования, дополняющую функциональный подход.

Что общего имеет React или Flux с функциональным JavaScript? Прежде всего, функциональный компонент, не имеющий состояния, является чистой функцией. Он получает инструкции в виде свойств и возвращает элементы пользовательского

интерфейса. Класс React использует состояние или свойства в качестве входных данных и также будет создавать элементы пользовательского интерфейса. Из компонентов React составляется единый компонент. Неизменяемые данные предоставляют компоненту входную информацию, а на выходе возвращаются элементы пользовательского интерфейса:

```
const Countdown = ({count}) => <h1>{count}</h1>
```

Flux предоставляет способ построения веб-приложений, дополняющий работу React. Если конкретнее, то Flux дает способ предоставления данных, которые React будет использовать для создания пользовательского интерфейса.

Во Flux данные состояния приложения управляются за пределами компонентов React в *хранилищах* (stores), в которых они содержатся и изменяются и которым дано исключительное право обновления Flux-представления. Если пользователь должен взаимодействовать с веб-страницей, например нажимать кнопку или отправлять данные формы, то для представления пользовательского запроса будет создано *действие* (action). Оно предоставляет инструкции и данные, требуемые для внесения изменений. Инструкции отправляются с помощью центрального управляющего элемента под названием «диспетчер» (dispatcher). Он сконструирован с целью выстраивания действий в очередь и отправки их в соответствующее хранилище. При получении действия хранилище использует инструкции для внесения изменений в состояние и обновление представления. Получается односторонний поток данных: действия проходят к диспетчеру, оттуда к хранилищу и, наконец, к представлению (рис. 7.7).

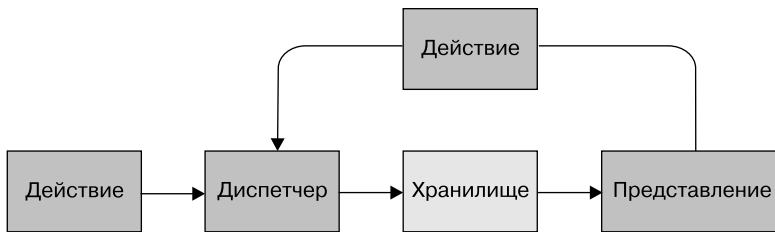


Рис. 7.7. Модель конструирования Flux, разработанная в Facebook

Во Flux действия и данные состояния являются неизменяемыми. Диспетчеризация действий может производиться из представления, или же они могут поступать из других источников, обычно с веб-сервера.

Для каждого изменения требуется действие. Каждое из них предоставляет инструкции для внесения изменений. Действия также служат в качестве квитанций, сообщающих о том, что изменилось, какие данные были задействованы для изменений и что явилось источником действия. Эта модель не вызывает побочных эффектов. Только хранилище может вносить изменения. Хранилище обновляет данные, представление отображает эти изменения в пользовательском интерфейсе, а действия сообщают, как и почему произошли изменения.

Сведение потока данных вашего приложения к этой модели конструирования существенно упростит поиск ошибок в приложении и его масштабирование. Посмотрите на приложение на рис. 7.8. Можно заметить, что каждое прошедшее через диспетчер действие было выведено в консоль. Эти действия сообщают о том, каким образом получен текущий пользовательский интерфейс, отображающий огромную цифру 3.

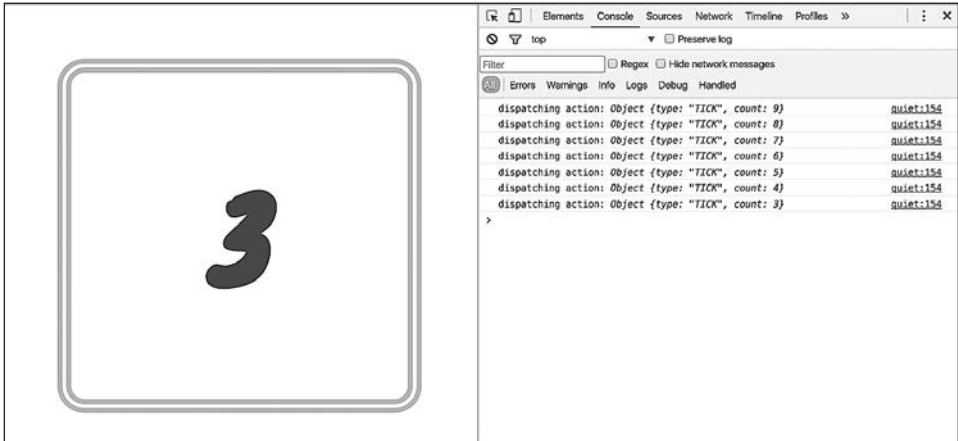


Рис. 7.8. Приложение обратного отсчета, созданное с применением Flux

Как видите, последнее изменение было связано с действием, имеющим тип `TICK`. Этим действием значение счетчика из предыдущего значения, которое, судя по рисунку, равнялось 4, было изменено на 3. Действия сообщают нам, как изменялось состояние. Их можно отследить в обратном порядке до источника и увидеть: первое изменение показания счетчика было с 10 на 9, следовательно, можно предположить, что это приложение ведет обратный отсчет с 10.

Посмотрим, как этот обратный отсчет создан с помощью шаблона конструирования Flux. Мы представим каждую часть данного шаблона и рассмотрим ее вклад в однонаправленный поток данных, из которого получается описанный обратный отсчет.

Представление

Начнем с представления, являющегося компонентом React, не имеющим состояния. Состояние приложения будет управляться с применением Flux, поэтому, пока не возникнет потребность в функции жизненного цикла, можно обойтись без компонентов, создаваемых с помощью классов.

Показание счетчика, предназначенное для вывода на экран, представление обратного отсчета получает в виде свойства. Оно также получает две функции, `tick` и `reset`:

```
const Countdown = ({count, tick, reset}) => {
  if (count) {
    setTimeout(() => tick(), 1000)
  }

  return (count) ?
    <h1>{count}</h1> :
    <div onClick={() => reset(10)}>
      <span>CELEBRATE!!!</span>
      <span>(click to start over)</span>
    </div>
}
```

При отображении этого представления на экран будет выведено показание счетчика, только если оно не будет нулевым; тогда пользователю будет выведено сообщение с призывом начинать праздновать — CELEBRATE!!!. Если показание счетчика отличается от нуля, то представление устанавливает паузу, ждет одну секунду и вызывает действие, имеющее тип TICK.

Когда показание счетчика становится нулевым, представление не станет вызывать каких-либо других создателей действий, пока пользователь не щелкнет на содержимом основного контейнера `div`, инициировав перезапуск. Тогда произойдет переустановка счетчика на значение 10, и процесс обратного отсчета начнется заново.



Состояние в компонентах

Использование Flux не означает, что в каком-либо из ваших компонентов представления не может быть состояния. Оно говорит о том, что состояние приложения не управляется в ваших компонентах представления. Например, Flux может управлять значениями дат и времени на шкале времени. Но применять для визуализации шкалы времени вашего приложения компоненты этой шкалы, имеющие внутреннее состояние, не запрещается.

Состояние нужно задействовать рационально, только в случае надобности, из повторно используемых компонентов, управляющих своим собственным состоянием внутри самих себя. Остальная часть приложения не должна быть «осведомлена» о состоянии дочернего компонента.

Действия и создатели действий

Действия предоставляют инструкции и данные, которые хранилище будет использовать для изменения состояния. *Создатели действий* представляют собой функции, благодаря которым можно абстрагировать всевозможные детали, необходимые для создания действия. Сами по себе действия — это объекты, как минимум

содержащие поле `type`. Тип действия обычно является строкой с символами в верхнем регистре, содержащей описание действия. Кроме того, в действия могут быть упакованы любые данные, которых требует хранилище. Например:

```
const countdownActions = dispatcher =>
  ({
    tick(currentCount) {
      dispatcher.handleAction({ type: 'TICK' })
    },
    reset(count) {
      dispatcher.handleAction({
        type: 'RESET',
        count
      })
    }
  })
})
```

Когда загружаются создатели действия обратного отсчета, им в виде аргумента отправляется диспетчер. При каждом инициировании действия `TICK` или `RESET` вызывается принадлежащий диспетчеру метод `handleAction`, который «выполняет диспетчеризацию» объекта действия.

Диспетчер

Существует только один диспетчер, и он является организатором перемещения данных этого шаблона проектирования. Диспетчер получает действие, упаковывает в него информацию о том, где оно было создано, и отправляет его в соответствующее хранилище или хранилища, которые займутся выполнением действия.

Хотя Flux не является фреймворком, Facebook открывает класс `Dispatcher`, имеющий открытый код, которым вы можете воспользоваться. Диспетчеры, как правило, имеют стандартную реализацию, поэтому лучше применять диспетчер Facebook, а не создавать собственную конструкцию:

```
import Dispatcher from 'flux'

class CountdownDispatcher extends Dispatcher {
  handleAction(action) {
    console.log('dispatching action:', action)
    this.dispatch({
      source: 'VIEW_ACTION',
      action
    })
  }
}
```

Когда метод `handleAction` вызывается с действием, это действие отправляется диспетчером вместе с данными о том, откуда оно взялось. Когда создается храни-

лице, оно регистрируется диспетчером и начинает отслеживать действия. При диспетчеризации действие обрабатывается в порядке его получения и отправляется в соответствующие хранилища.

Хранилища

Хранилища представляют собой объекты, в которых содержатся логика и данные состояния приложения. Хранилища похожи на модели в шаблоне MVC, но не ограничиваются управлением данными в одном объекте. Можно создавать Flux-приложения, состоящие из одного хранилища, которое управляет данными, имеющими множество разнообразных типов.

Текущие данные состояния могут быть получены из хранилища через свойства. Все нужное хранилищу для изменения данных состояния предоставляется в действиях. Хранилище будет обрабатывать действия по их типу и соответственным образом изменять их данные. После изменения данных хранилище выдаст событие и уведомит любые представления, подписавшиеся на хранилище, что его данные изменились. Рассмотрим пример:

```
import { EventEmitter } from 'events'

class CountdownStore extends EventEmitter {

  constructor(count=5, dispatcher) {
    super()
    this._count = count
    this.dispatcherIndex = dispatcher.register(
      this.dispatch.bind(this)
    )
  }

  get count() {
    return this._count
  }

  dispatch(payload) {
    const { type, count } = payload.action
    switch(type) {
      case "TICK":
        this._count = this._count - 1
        this.emit("TICK", this._count)
        return true
      case "RESET":
        this._count = count
        this.emit("RESET", this._count)
        return true
    }
  }
}
```

В этом хранилище находится состояние приложения обратного отсчета, то есть показание счетчика. Доступ к состоянию показания счетчика можно получить через свойство, предназначенное только для чтения. Когда действия прошли диспетчеризацию, хранилище применяет их, чтобы изменить показания счетчика. Действие `TICK` уменьшает его показание на единицу. А действие `RESET` переустанавливает показание целиком, используя данные, включенные в действие.

Как только состояние изменилось, хранилище выдает событие любому представлению, которое может его отслеживать.

А теперь все вместе

Мы узнали, как поток данных проходит через каждую часть Flux-приложения. Теперь посмотрим, как все эти части присоединяются друг к другу:

```
const appDispatcher = new CountdownDispatcher()
const actions = countdownActions(appDispatcher)
const store = new CountdownStore(10, appDispatcher)

const render = count => ReactDOM.render(
  <Countdown count={count} {...actions} />,
  document.getElementById('react-container')
)

store.on("TICK", () => render(store.count))
store.on("RESET", () => render(store.count))
render(store.count)
```

Сначала создается `appDispatcher`. Затем `appDispatcher` используется для генерирования создателей наших действий. И наконец, в `appDispatcher` регистрируется наше хранилище и устанавливает в качестве начального показания счетчика число `10`.

Метод `render` применяется для отображения представления с показанием счетчика, которое он получает в виде аргумента. Он также отправляет представлению в виде свойств создатели действий.

В завершение к хранилищу добавляются слушатели, замыкающие цикл. Когда хранилище выдает `TICK` или `RESET`, оно также выдает новое показание счетчика, которое тут же отображается в представлении. Затем исходное представление выводится на экран с показанием счетчика, имеющимся в хранилище. Всякий раз, когда представление выдает `TICK` или `RESET`, действие отправляется по этому циклу и в конечном итоге возвращается представлению в виде данных, готовых к отображению.

Реализации Flux

Для реализации Flux существуют различные подходы. Конкретные реализации этого шаблона проектирования можно найти в нескольких библиотеках с открытым кодом:

- ❑ *Flux* — принадлежащая Facebook библиотека Flux представляет собой только что рассмотренный нами шаблон проектирования; включает реализацию диспетчера;
- ❑ *Reflux* — упрощенный подход к однонаправленному потоку данных, который сконцентрирован на действиях, хранилищах и представлениях;
- ❑ *Flummox* — реализация Flux, позволяющая создавать Flux-модули с помощью расширения классов JavaScript;
- ❑ *Fluxible* — фреймворк Flux, созданный компанией Yahoo! для работы с изоморфными Flux-приложениями (будут рассматриваться в главе 12);
- ❑ *Redux* — Flux-подобная библиотека, в которой модульность достигается за счет использования функций, а не объектов;
- ❑ *MobX* — библиотека управления состоянием, в которой для реагирования на изменения состояния применяются наблюдаемые объекты.

У всех этих реализаций имеются механизмы хранилищ, действий и диспетчеров, а в качестве уровня представления используются компоненты React. Все они являются вариациями шаблона проектирования Flux, в основу которого положен однонаправленный поток данных.

Одной из вариаций фреймворка Flux, быстро набирающего весьма широкую популярность, является Redux. В следующей главе мы обсудим, как использовать Redux, чтобы конструировать архитектуру функциональных данных для ваших приложений на стороне клиента.

8

Redux

Библиотека Redux (<http://redux.js.org/>) вошла в число явных победителей среди Flux-библиотек, или библиотек со сходными задачами. Она основана на модели Flux и была разработана для решения проблемы понимания того, как именно изменяются данные, проходя потоком через компоненты вашего приложения. Redux разработали Дэн Абрамов (Dan Abramov) (<https://github.com/gaearon>) и Эндрю Кларк (Andrew Clark). После создания Redux оба они были наняты компанией Facebook для работы в составе команды React.

Эндрю Кларк работал над версией 4 Flummox, еще одной библиотеки, основанной на модели Flux, когда стал помогать Дэну закончить работу с Redux. В сообщении на npm-странице для Flummox (<https://www.npmjs.com/package/flummox>) говорится следующее.

«Со временем выпуск 4.x должен был стать последним из основных выпусков, но этому не суждено будет случиться. Если вам нужны самые свежие решения, воспользуйтесь вместо него библиотекой Redux. Она действительно хороша»¹.

Библиотека Redux удивительно мала по объему, в ней всего 99 строк кода (<https://gist.github.com/gaearon/ffd88b0e4f00b22c3159>).

Мы говорили, что Redux является Flux-подобной библиотекой, но это не совсем Flux. В ней есть действия, создатели действий, хранилище и объекты действий, используемые для изменения состояния. Redux немного упрощает концепции Flux за счет удаления диспетчера и представления состояния приложения с помощью единственного неизменяемого объекта. В Redux также введены *преобразователи* (reducers), не являющиеся составной частью модели Flux. Преобразователи представляют собой чистые функции, возвращающие новое состояние на основе текущего состояния и действия: `(state, action) => newState`.

¹ Документация Flummox. <https://github.com/acdlite/flummox>.

Состояние

Идея хранения состояния в одном месте не такая уж и бредовая. Фактически в последней главе мы делали то же самое, сохраняя состояние в корне нашего приложения. В чистых приложениях React или Flux рекомендуется хранить состояние в как можно меньшем количестве объектов. А для Redux это правило¹.

Требование хранить состояние в одном месте может показаться неразумным, особенно когда работа ведется с различными типами данных. Посмотрим, как этого можно добиться в приложении, имеющем множество различных типов данных. Рассмотрим приложение для социальных сетей, у которого состояние разбросано по разным компонентам (рис. 8.1). У самого приложения имеется состояние пользователя. Под этим состоянием хранятся все сообщения. У каждого сообщения (message) имеется свое собственное состояние, а все публикации (posts) хранятся под компонентом публикаций.

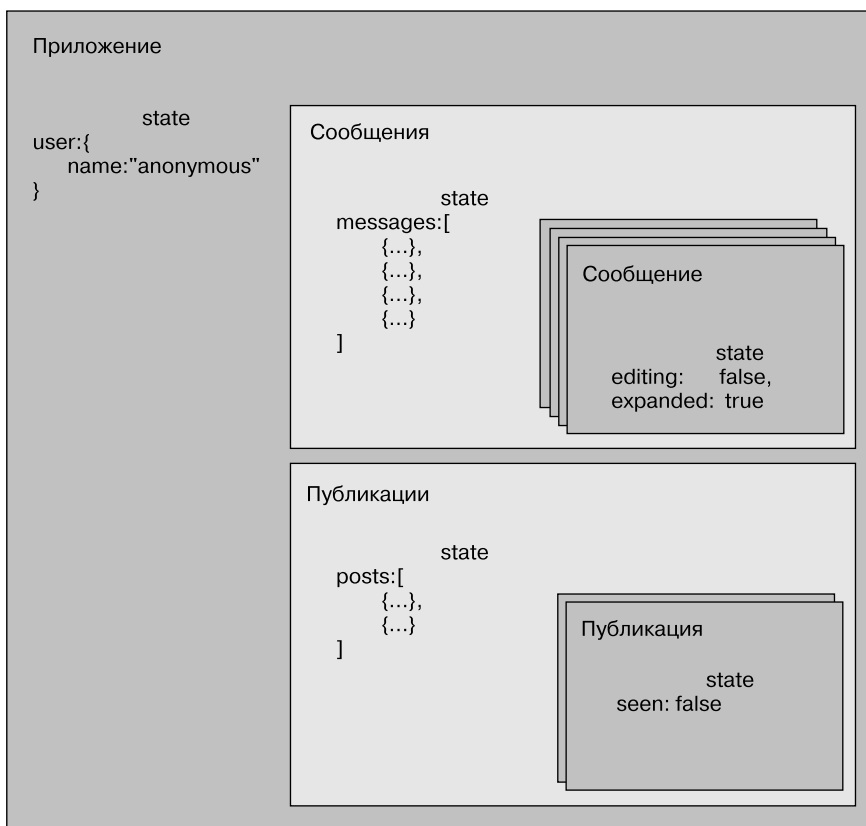


Рис. 8.1. React-приложение, в котором компоненты хранят собственные состояния

¹ Документация Redux: Three Principles. <http://redux.js.org/docs/introduction/ThreePrinciples.html>.

Приложения, имеющие подобную структуру, способны работать вполне успешно, но по мере роста их объема может быть все труднее определить общее состояние приложения. Кроме того, будет довольно трудно разобраться, откуда поступают обновления, при условии, что каждый компонент станет изменять свое собственное состояние внутренними вызовами метода `setState`.

Какие сообщения были дополнены? Какие публикации были прочитаны? Чтобы выяснить все эти подробности, нужно углубиться в дерево компонентов и отследить состояния внутри отдельно взятых компонентов.

Redux упрощает способ просмотра состояния в приложении, требуя от нас хранить все данные состояния в одном объекте. Все, что нужно знать о приложении, находится в одном месте: единственном источнике истины. Точно такое же приложение можно сконструировать с Redux, переместив все данные состояния в одно место (рис. 8.2).

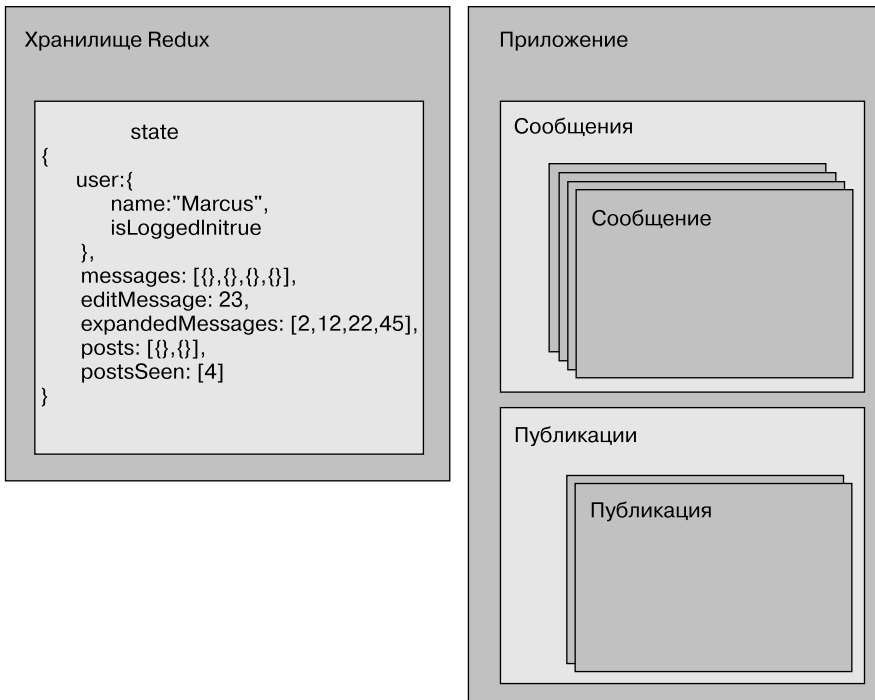


Рис. 8.2. Redux требует, чтобы все состояние хранилось в одном неизменяемом объекте

В приложении социальной сети можно увидеть, что управление состоянием текущего пользователя, сообщений и публикаций ведется из одного и того же объекта: хранилища Redux. В нем даже хранится информация об отредактированных и дополненных сообщениях, просмотренных публикациях. Эта информация собрана в массивах, содержащих идентификаторы, ссылающиеся на конкретные записи.

В этом объекте состояния кэшируются все сообщения и публикации, так что данные находятся под рукой.

При использовании Redux все управление состоянием полностью удаляется из React. Состоянием будет управлять Redux.

Дерево состояния для приложения социальной сети показано на рис. 8.3. В нем имеются сообщения в массиве. То же самое справедливо и для публикаций. Все, что нам нужно, берет свое начало в одном объекте: дереве состояния. Каждый ключ в одном и том же объекте представляет ветвь дерева состояния.

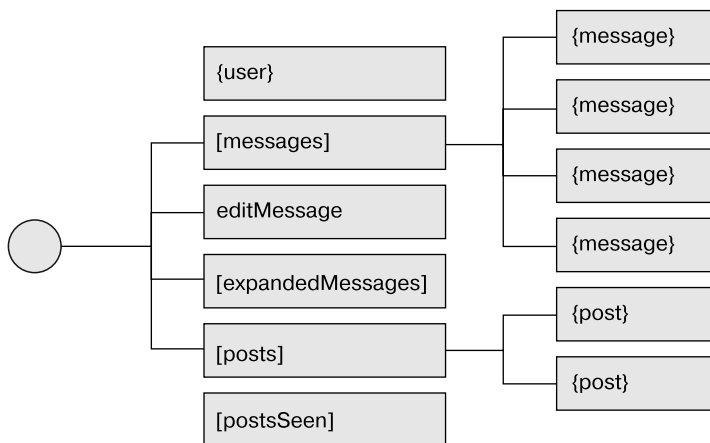


Рис. 8.3. Пример дерева состояния

При создании Redux-приложений состояние является первым, о чем нужно подумать. Постарайтесь определить его в одном объекте. Обычно рекомендуется составлять черновой JSON-проект вашего дерева состояния с местами, предназначенными для заполнения данными.

Вернемся к нашему приложению организера цветов. В этом приложении у нас есть информация о каждом цветке, хранящаяся в массиве, и информация о том, как эти цветы должны быть отсортированы. Образец данных состояния будет похож на тот, что показан в примере 8.1.

Пример 8.1. Примерное состояние приложения организера цветов

```

{
  colors: [
    {
      "id": "8658c1d0-9eda-4a90-95e1-8001e8eb6036",
      "title": "Ocean Blue",
      "color": "#0070ff",
      "rating": 3,
      "timestamp": "Sat Mar 12 2016 16:12:09 GMT-0800 (PST)"
    }
  ]
}

```

```

    },
    {
      "id": "f9005b4e-975e-433d-a646-79df172e1dbb",
      "title": "Tomato",
      "color": "#d10012",
      "rating": 2,
      "timestamp": "Fri Mar 11 2016 12:00:00 GMT-0800 (PST)"
    },
    {
      "id": "58d9caee-6ea6-4d7b-9984-65b145031979",
      "title": "Lawn",
      "color": "#67bf4f",
      "rating": 1,
      "timestamp": "Thu Mar 10 2016 01:11:12 GMT-0800 (PST)"
    },
    {
      "id": "a5685c39-6bdc-4727-9188-6c9a00bf7f95",
      "title": "Party Pink",
      "color": "#ff00f7",
      "rating": 5,
      "timestamp": "Wed Mar 9 2016 03:26:00 GMT-0800 (PST)"
    }
  ],
  sort: "SORTED_BY_DATE"
}

```

После того как мы разобрались с основной структурой состояния нашего приложения, посмотрим, как это состояние обновляется и изменяется с помощью действий.

Действия

В последнем разделе было введено важное правило Redux: состояние приложения должно храниться в одном неизменяемом объекте. *Неизменяемость* означает, что этот объект состояния не должен изменяться. Время от времени он будет обновляться за счет его полной замены. Чтобы воплотить данный замысел в жизнь, нам понадобятся инструкции, касающиеся изменений. Именно это и предоставляют *действия*: инструкции, касающиеся изменений, вносимых в состояние приложения, которые сопровождаются данными, необходимыми для внесения изменений¹.

Действия являются единственным способом обновления состояния Redux-приложения. Они предоставляют нам инструкции о том, что должно быть изменено, но мы можем рассматривать их и в качестве записей истории изменений, произошедших со временем. Если пользователи удалили три цвета, добавили четыре, а затем оценили пять, в результате останутся информационные следы, показанные на рис. 8.4.

¹ Документация Redux: Actions. <http://redux.js.org/docs/basics/Actions.html>.

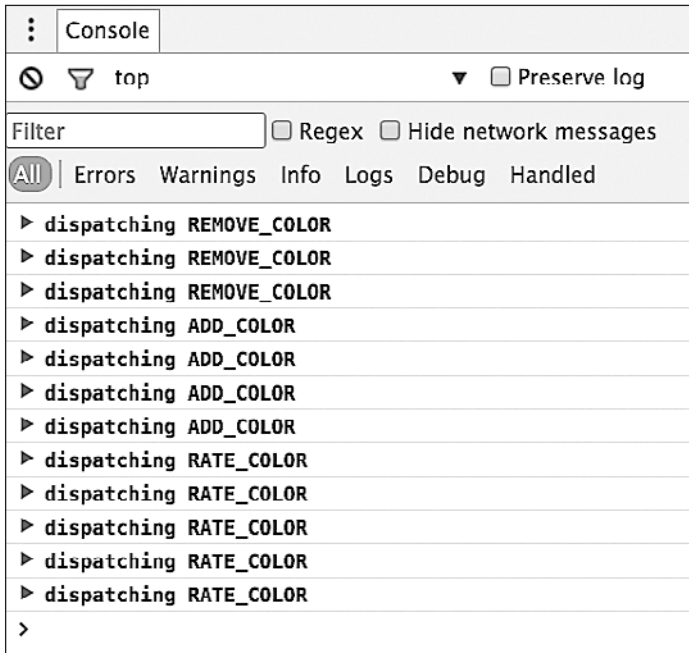


Рис. 8.4. Действия, информация о которых выведена на консоль в процессе их диспетчеризации

Обычно когда мы беремся за конструирование объектно-ориентированного приложения, работа начинается с выявления объектов, их свойств и порядка их совместной работы. Наше мышление в данном случае *сконцентрировано на названиях*. При создании Redux-приложения хотелось бы сместить мышление в сторону *концентрации на действиях*. Как действия повлияют на данные состояния? После выявления действий их можно перечислить в файле `constants.js` (пример 8.2).

Пример 8.2. Константы, перечисленные в файле `./constants.js`

```
const constants = {
  SORT_COLORS: "SORT_COLORS",
  ADD_COLOR: "ADD_COLOR",
  RATE_COLOR: "RATE_COLOR",
  REMOVE_COLOR: "REMOVE_COLOR"
}
export default constants
```

При работе с органайзером цветов пользователям понадобится добавлять цвета, оценивать их, удалять или выполнять сортировку списка. Здесь определены строковые значения для каждого из этих типов действий. Действие представляет собой объект JavaScript, имеющий как минимум поле для типа:

```
{ type: "ADD_COLOR" }
```

Тип действия является строкой, определяющей то, что должно произойти. `ADD_COLOR` представляет действие добавления нового цвета к списку в состоянии приложения.

При создании действий с использованием строк вполне возможно допустить опечатку:

```
{ type: "ADD_COOLOR" }
```

Эта опечатка превратится в ошибку в приложении. Ошибки подобного рода зачастую не приводят к выдаче предупреждений; вы просто не увидите в данных состояния ожидаемых изменений. Такие ошибки порой довольно трудно обнаруживаются. И здесь на помощь могут прийти *константы*:

```
import C from "./constants"  
  
{ type: C.ADD_COLOR }
```

Этот код указывает на то же самое действие, но с помощью константы JavaScript, а не строки. Опечатка в названии переменной JavaScript приведет к тому, что браузер выдаст ошибку. Определение действия в виде констант также позволит воспользоваться в вашей IDE преимуществами, предоставляемыми средствами автодополнения и автозавершения, например таким средством, как IntelliSense. Определение действия в виде констант также позволит воспользоваться в вашей IDE преимуществами. При наборе первых одной-двух букв константы IDE автоматически завершит ее ввод за вас. Использование констант не является обязательным требованием, но было бы вполне разумно выработать в себе привычку к их применению.



Соглашения по именам действий

Типы действий, такие как `ADD_COLOR` или `RATE_COLOR`, являются всего лишь строками, следовательно, с технической точки зрения действия можно называть как угодно. Обычно типы действий прописываются заглавными буквами со знаками подчеркивания вместо пробелов. Нужно также стремиться четко формулировать предназначение действия.

Целевые данные действия. Действия — литералы JavaScript, предоставляющие инструкции, необходимые для внесения изменений в состояние. Большинству изменений состояния также нужны данные. Какие именно записи следует удалить? Какую новую информацию нужно предоставить в новой записи?

На такие данные мы ссылаемся как на целевые данные действия. Например, при диспетчеризации такого действия, как `RATE_COLOR`, нужно будет знать, какой цвет оценивать и какое показание рейтинга применить к нему. Эта информация может быть передана непосредственно с действием в том же самом литерале JavaScript (пример 8.3).

Пример 8.3. Действие RATE_COLOR

```
{
  type: "RATE_COLOR",
  id: "a5685c39-6bdc-4727-9188-6c9a00bf7f95",
  rating: 4
}
```

Пример 8.3 содержит тип действия RATE_COLOR и данные, необходимые для изменения рейтинга указанного цвета на 4.

При добавлении новых цветов понадобится информация о добавляемом цвете (пример 8.4).

Пример 8.4. Действие ADD_COLOR

```
{
  type: "ADD_COLOR",
  color: "#FFFFFF",
  title: "Bright White",
  rating: 0,
  id: "b5685c39-3bdc-4727-9188-6c9a33df7f52",
  timestamp: "Sat Mar 12 2016 16:12:09 GMT-0800 (PST)"
}
```

Это действие предписывает Redux добавить к состоянию новый цвет под названием **Bright white**. Все информация о нем включена в действие. Действия представлены в виде компактных пакетов, сообщающих Redux, как должно измениться состояние. В них также включены все данные, которые понадобятся Redux для внесения изменения.

Преобразователи

Все наше дерево состояния хранится в одном объекте. Возможно, кто-то выразит недовольство из-за недостаточной модульности, если рассматривать ее в виде описания объектов. В Redux модульность достигается за счет функций. Они используются для обновления частей дерева состояния. Эти функции называются *преобразователями* (reducers)¹.

Преобразователи представляют собой функции, которые получают текущее состояние и действие в виде аргументов и используют их для создания и возвращения нового состояния. Разрабатываются для обновления конкретных частей дерева состояния: либо листьев, либо ветвей. Затем преобразователи можно собирать в один, управляющий всем состоянием приложения при любых действиях.

Органайзер цветов хранит все данные состояния в одном дереве (пример 8.5). Если для этого приложения потребуется использовать Redux, можно создать несколько преобразователей, каждый из которых нацелен на конкретные листья или ветви дерева состояния.

¹ Документация Redux: Reducers. <http://redux.js.org/docs/basics/Reducers.html>.

Пример 8.5. Состояние приложения органайзера цветов

```

{
  colors: [
    {
      "id": "8658c1d0-9eda-4a90-95e1-8001e8eb6036",
      "title": "Ocean Blue",
      "color": "#0070ff",
      "rating": 3,
      "timestamp": "Sat Mar 12 2016 16:12:09 GMT-0800 (PST)"
    },
    {
      "id": "f9005b4e-975e-433d-a646-79df172e1dbb",
      "title": "Tomato",
      "color": "#d10012",
      "rating": 2,
      "timestamp": "Fri Mar 11 2016 12:00:00 GMT-0800 (PST)"
    },
    {
      "id": "58d9caee-6ea6-4d7b-9984-65b145031979",
      "title": "Lawn",
      "color": "#67bf4f",
      "rating": 1,
      "timestamp": "Thu Mar 10 2016 01:11:12 GMT-0800 (PST)"
    },
    {
      "id": "a5685c39-6bdc-4727-9188-6c9a00bf7f95",
      "title": "Party Pink",
      "color": "#ff00f7",
      "rating": 5,
      "timestamp": "Wed Mar 9 2016 03:26:00 GMT-0800 (PST)"
    }
  ],
  sort: "SORTED_BY_DATE"
}

```

У этих данных состояния есть две основные ветви: `colors` и `sort`. Последняя, по сути, является листом, поскольку не содержит никаких дочерних узлов. В ветви `colors` хранятся несколько цветов. Каждый объект `color` представлен листом (рис. 8.5).

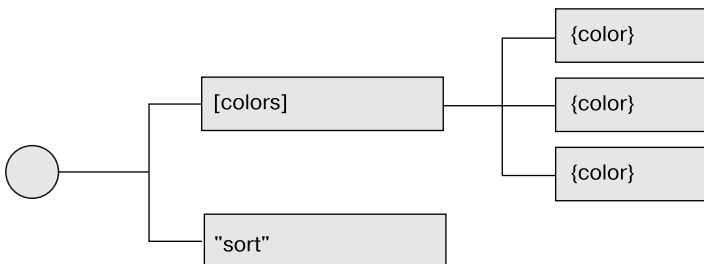


Рис. 8.5. Дерево состояния органайзера цветов

Для работы с каждой частью данного дерева состояния будет использоваться отдельный преобразователь. Каждый из них является простой функцией, поэтому вместо всех преобразователей можно сразу создать имитаторы, код которых показан в примере 8.6.

Пример 8.6. Имитация преобразователей в органайзере цветов

```
import C from '../constants'

export const color = (state={}, action) => {
  return {}
}

export const colors = (state=[], action) => {
  return []
}

export const sort = (state="SORTED_BY_DATE", action) => {
  return ""
}
```

Обратите внимание: преобразователь цвета должен быть объектом и возвращает объект. Этот преобразователь получает состояние в виде массива и возвращает массив. Преобразователь сортировки получает строку и возвращает строку. Каждая функция нацелена на конкретную часть дерева состояния. Возвращаемое значение и исходное состояние для каждой функции соответствуют своим типам данных в дереве. Цвета хранятся в массиве. Каждый цвет — объект. Свойство `sort` является строкой.

Каждый преобразователь предназначен для обработки только тех действий, которые необходимы для обновления его части дерева состояния. Преобразователь цвета станет обрабатывать только действия, необходимые для нового или измененного объекта цвета: `ADD_COLOR` и `RATE_COLOR`. Преобразователь цветов будет нацелен на действия, необходимые для управления массивом `colors`: `ADD_COLOR`, `REMOVE_COLOR`, `RATE_COLOR`. И наконец, преобразователь сортировки предназначен для обработки действия `SORT_COLORS`.

Каждый преобразователь состоит из функции или сводится в одну функцию преобразователя, которая будет использовать хранилище. Преобразователь цветов состоит из преобразователя цвета, предназначенного для управления отдельными цветами в массиве. Затем для создания одной функции преобразователя с преобразователем цветов будет объединен преобразователь сортировки. Он сможет обновлять сразу все дерево состояния и обрабатывать любое отправленное ему действие (рис. 8.6).

Оба преобразователя, и цвета и цветов, будут обрабатывать действия `ADD_COLOR` и `RATE_COLOR`. Но нужно помнить, что каждый преобразователь нацелен на конкретную часть дерева состояния. Действие `RATE_COLOR` в преобразователе цвета будет обрабатывать задачу изменения значения рейтинга отдельно взятого образца.

А действие того же типа, `RATE_COLOR`, в преобразователе цветов будет нацелено на местоположение в массиве цвета, который должен быть оценен. Действие `ADD_COLOR` в преобразователе цвета приведет к созданию нового объекта цвета с соответствующими свойствами. Действие того же типа, `ADD_COLOR`, в преобразователе цветов вызовет возвращение массива, имеющего дополнительный объект цвета. Они разработаны для совместной работы. Каждый преобразователь нацелен на реализацию действия, предназначенного именно для его ветви в дереве состояния.

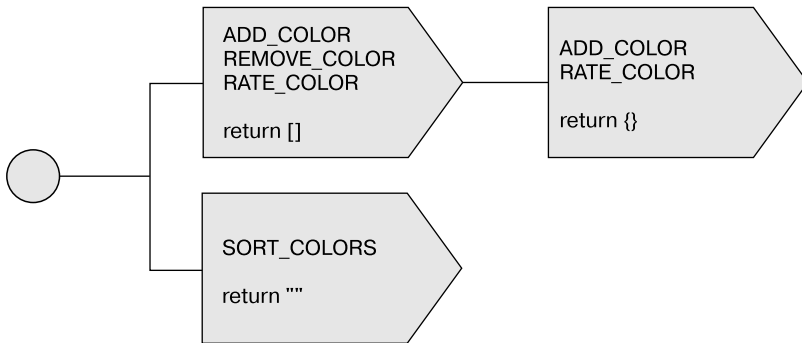


Рис. 8.6. Дерево преобразователей организера цветов



Объединение преобразователей не требование, а просто рекомендация

Redux не требует создавать меньшие по объему и более целенаправленные преобразователи и объединять их в один. В приложении можно создать одну функцию преобразователя для обработки всего действия. Но, поступая таким образом, мы утратим преимущества модульности и функционального программирования.

Преобразователь цвета

Преобразователи можно запрограммировать несколькими различными способами. Весьма популярным решением является использование инструкций `switch`, поскольку с их помощью можно обрабатывать различные типы действий, с которыми должен работать преобразователь. Он проверяет тип действия `action.type` в инструкции `switch`, а затем обрабатывает каждый тип действия в варианте `case` инструкции `switch`:

```
export const color = (state = {}, action) => {
  switch (action.type) {
    case C.ADD_COLOR:
```

```

    return {
      id: action.id,
      title: action.title,
      color: action.color,
      timestamp: action.timestamp,
      rating: 0
    }
  case C.RATE_COLOR:
    return (state.id !== action.id) ?
      state :
      {
        ...state,
        rating: action.rating
      }
  default :
    return state
}
}
}

```

Действиями для преобразователя цвета являются:

- ❑ `ADD_COLOR` — возвращает объект цвета, сконструированный из целевых данных действия;
- ❑ `RATE_COLOR` — возвращает новый объект цвета с желаемым рейтингом; оператор распространения ES7 позволяет присвоить новому объекту значение текущего состояния.

Преобразователи всегда должны что-нибудь возвращать. Если по какой-либо причине данный преобразователь был вызван с неопознанным действием, то как вариант по умолчанию (`default`) им будет возвращено текущее состояние.

Теперь, располагая преобразователем цвета, мы можем воспользоваться им, чтобы вернуть новые цвета или рейтинг существующих образцов. Например:

```

// Добавление нового цвета
const action = {
  type: "ADD_COLOR",
  id: "4243e1p0-9abl-4e90-95p4-800118yf3036",
  color: "#0000FF",
  title: "Big Blue",
  timestamp: "Thu Mar 10 2016 01:11:12 GMT-0800 (PST)"
}

console.log( color({}, action) )

// Вывод в консоль
// {
//   id: "4243e1p0-9abl-4e90-95p4-800118yf3036",
//   color: "#0000FF",
//   title: "Big Blue",
//   timestamp: "Thu Mar 10 2016 01:11:12 GMT-0800 (PST)",
//   rating: "0"
// }

```

Новый объект цвета возвращается со всеми представленными полями, включая рейтинг по умолчанию, равный нулю. Чтобы внести изменения в существующий цвет, можно послать действие `RATE_COLOR` с идентификатором и новым значением рейтинга:

```
const existingColor = {
  id: "128e1p5-3abl-0e52-33p0-840118yf3036",
  title: "Big Blue",
  color: "#0000FF",
  timestamp: "Thu Mar 10 2016 01:11:12 GMT-0800 (PST)",
  rating: 0
}

const action = {
  type: "RATE_COLOR",
  id: "4243e1p0-9abl-4e90-95p4-800118yf3036",
  rating: 4
}

console.log( color(existingColor, action) )

// Вывод на консоль
// {
//   id: "4243e1p0-9abl-4e90-95p4-800118yf3036",
//   title: "Big Blue",
//   color: "#0000FF",
//   timestamp: "Thu Mar 10 2016 01:11:12 GMT-0800 (PST)",
//   rating: 4
// }
```

Преобразователь цвета — функция, которая создает новый объект или устанавливает новый рейтинг для существующего цвета. Обратите внимание: действие `RATE_COLOR` передает идентификатор, не используемый преобразователем цвета. Дело в том, что ID этого действия применяется с целью задать местоположение цвета в совершенно другом преобразователе. Объект действия может влиять на несколько преобразователей.

Преобразователь цветов

Преобразователь цвета разработан для управления листьями цветовой ветви дерева состояния. А преобразователь цветов будет использоваться для управления всей цветовой ветвью:

```
export const colors = (state = [], action) => {
  switch (action.type) {
    case C.ADD_COLOR :
      return [
        ...state,
        color({}, action)
      ]
    case C.RATE_COLOR :
```

```

    return state.map(
      c => color(c, action)
    )
  case C.REMOVE_COLOR :
    return state.filter(
      c => c.id !== action.id
    )
  default:
    return state
}
}

```

Этот преобразователь будет обрабатывать любые действия по добавлению, оценке и удалению цветов.

- ❑ **ADD_COLOR** — создает новый массив путем объединения всех значений существующего массива состояния с новым объектом цвета. Новый цвет создается с помощью передачи пустого объекта состояния и действия преобразователю цвета.
- ❑ **RATE_COLOR** — возвращает новый массив цветов с желаемым рейтингом цвета. Преобразователь цветов определяет местоположение оцениваемого цвета в текущем массиве состояния. Затем использует преобразователь цвета для получения только что оцененного объекта цвета и выполняет его замену в массиве.
- ❑ **REMOVE_COLOR** — создает новый массив, применяя фильтр к заданному цвету для удаления этого образца.

Преобразователь образцов цвета работает с массивом цветов. Чтобы сфокусироваться на отдельных объектах, его использует преобразователь цвета.



Подход к состоянию как к неизменяемому объекту

Во всех этих преобразователях к состоянию нужно подходить как к неизменяемому объекту. Следует удерживаться от соблазна применять такие выражения, как `state.push({})` или `state[index].rating`.

Теперь цвета можно добавлять, оценивать или удалять из массива с помощью следующей чистой функции:

```

const currentColors = [
  {
    id: "9813e2p4-3ab1-2e44-95p4-800118yf3036",
    title: "Berry Blue",
    color: "#000066",
    rating: 0,
    timestamp: "Thu Mar 10 2016 01:11:12 GMT-0800 (PST)"
  }
]

```

```
const action = {
  type: "ADD_COLOR",
  id: "5523e7p8-3ab2-1e35-95p4-800118yf3036",
  title: "Party Pink",
  color: "#F142FF",
  timestamp: "Thu Mar 10 2016 01:11:12 GMT-0800 (PST)"
}

console.log( colors(currentColors, action) )

// Вывод на консоль
// [{
//   id: "9813e2p4-3ab1-2e44-95p4-800118yf3036",
//   title: "Berry Blue",
//   color: "#000066",
//   timestamp: "Thu Mar 10 2016 01:11:12 GMT-0800 (PST)",
//   rating: 0
// },
// {
//   id: "5523e7p8-3ab2-1e35-95p4-800118yf3036",
//   title: "Party Pink",
//   color: "#F142FF",
//   timestamp: "Thu Mar 10 2016 01:11:12 GMT-0800 (PST)",
//   rating: 0
// }]
```



В преобразователях не должно быть никаких побочных эффектов

Преобразователи должны быть предсказуемыми. Они используются просто для управления данными состояния. Обратите внимание: в предыдущем примере метка времени и идентификаторы сгенерированы до отправки действия преобразователю. Генерирование случайных данных, вызов функций API и другие асинхронные процессы должны обрабатываться за пределами преобразователей. Для них неизменно рекомендуется избегать изменения состояния и побочных эффектов.

Кроме того, можно удалить цвет из состояния или оценить конкретный образец, отправив соответствующее действие преобразователю цветов.

Преобразователь сортировки

Преобразователь сортировки является целой функцией, разработанной для управления в состоянии одной строковой переменной:

```
export const sort = (state = "SORTED_BY_DATE", action) => {
  switch (action.type) {
    case C.SORT_COLORS:
```



```
        return action.sortBy
    default :
        return state
    }
}
```

Преобразователь сортировки используется для изменения переменной состояния `sort`. Он устанавливает для состояния сортировки значение имеющегося в действии поля `sortBy` (если это состояние не будет предоставлено, то преобразователь возвратит `SORTED_BY_DATE`):

```
const state = "SORTED_BY_DATE"

const action = {
  type: C.SORT_COLORS,
  sortBy: "SORTED_BY_TITLE"
}

console.log( sort(state, action) ) // "SORTED_BY_TITLE"
```

Итак, обновление состояния выполняется преобразователями, которые являются чистыми функциями, получающими в качестве первого аргумента состояние, а в качестве второго аргумента — действие. Преобразователи не вызывают побочных эффектов и должны рассматривать свои аргументы в качестве неизменяемых данных. Модульность в Redux достигается с помощью преобразователей. В конечном итоге преобразователи сводятся в один, то есть в функцию, которая может обновить все дерево состояния.

В данном разделе мы показали, как можно скомпоновать преобразователи. Кроме того, описали, как преобразователь цветов использует преобразователь цвета, чтобы помочь в управлении цветами. В следующем разделе мы обсудим, как преобразователь цветов объединить с преобразователем сортировки с целью обновления состояния.

Хранилище

В Redux хранилищем считается то место, где хранятся данные состояния приложения и обрабатываются все обновления состояния¹. Хотя модель конструирования Flux допускает наличие множества хранилищ, каждое из которых нацелено на конкретный набор данных, в Redux имеется только одно.

Хранилище занимается обновлениями состояния, пропуская текущее состояние и действие через единый преобразователь. Мы создадим его путем сочетания и составления в него всех наших преобразователей.

¹ Документация Redux, Store. <http://redux.js.org/docs/basics/Store.html>.

Если создавать хранилище с помощью преобразователя цветов, то наш объект состояния будет массивом, то есть массивом цветов. Принадлежащий хранилищу метод `getState` станет возвращать настоящее состояние приложения. В примере 8.7 создается хранилище с преобразователем цвета; это является доказательством того, что для создания хранилища можно воспользоваться любым преобразователем.

Пример 8.7. Хранилище с преобразователем цвета

```
import { createStore } from 'redux'
import { color } from './reducers'

const store = createStore(color)

console.log( store.getState() ) // {}
```

Чтобы создать единое дерево преобразователей, похожее на изображенное на рис. 8.6 из предыдущего раздела, нужно составить комбинацию из преобразователей цветов и сортировки. В Redux для этого имеется специально предназначенная функция `combineReducers`, которая сводит все преобразователи в единый. Эти преобразователи используются для создания вашего дерева состояния. Имена полей соответствуют именам переданных преобразователей.

Хранилище также может быть создано с начальными данными. Вызов преобразователя цветов без состояния приведет к возвращению пустого массива:

```
import { createStore, combineReducers } from 'redux'
import { colors, sort } from './reducers'

const store = createStore(
  combineReducers({ colors, sort })
)

console.log( store.getState() )

// Вывод на консоль
//{
//  colors: [],
//  sort: "SORTED_BY_DATE"
//}
```

В примере 8.8 хранилище было создано с тремя цветами и значением сортировки `SORTED_BY_TITLE`.

Пример 8.8. Исходные данные состояния

```
import { createStore, combineReducers } from 'redux'
import { colors, sort } from './reducers'

const initialState = {
  colors: [
    {
```

```

    id: "3315e1p5-3abl-0p523-30e4-800118yf3036",
    title: "Rad Red",
    color: "#FF0000",
    rating: 3,
    timestamp: "Sat Mar 12 2016 16:12:09 GMT-0800 (PST)"
  },
  {
    id: "3315e1p5-3abl-0p523-30e4-800118yf4457",
    title: "Crazy Green",
    color: "#00FF00",
    rating: 0,
    timestamp: "Fri Mar 11 2016 12:00:00 GMT-0800 (PST)"
  },
  {
    id: "3315e1p5-3abl-0p523-30e4-800118yf2412",
    title: "Big Blue",
    color: "#0000FF",
    rating: 5,
    timestamp: "Thu Mar 10 2016 01:11:12 GMT-0800 (PST)"
  }
],
sort: "SORTED_BY_TITLE"
}

const store = createStore(
  combineReducers({ colors, sort }),
  initialState
)

console.log( store.getState().colors.length ) // 3
console.log( store.getState().sort ) // "SORTED_BY_TITLE"

```

Единственным способом изменения состояния вашего приложения является диспетчеризация действий через хранилище. В нем имеется метод `dispatch`, готовый получить действия в виде аргумента. При диспетчеризации с помощью хранилища действие проводится через преобразователи и состояние обновляется:

```

console.log(
  "Length of colors array before ADD_COLOR",
  store.getState().colors.length
)

// Длина массива colors перед действием ADD_COLOR равна 3

store.dispatch({
  type: "ADD_COLOR",
  id: "2222e1p5-3abl-0p523-30e4-800118yf2222",
  title: "Party Pink",
  color: "#F142FF",
  timestamp: "Thu Mar 10 2016 01:11:12 GMT-0800 (PST)"
})

console.log(
  "Length of colors array after ADD_COLOR",
  store.getState().colors.length
)

```

```
// Длина массива colors после действия ADD_COLOR равна 4
console.log(
  "Color rating before RATE_COLOR",
  store.getState().colors[3].rating
)
// Рейтинг цвета перед действием RATE_COLOR равен 0

store.dispatch({
  type: "RATE_COLOR",
  id: "2222e1p5-3abl-0p523-30e4-800118yf2222",
  rating: 5
})

console.log(
  "Color rating after RATE_COLOR",
  store.getState().colors[3].rating
)

// Рейтинг цвета после действия RATE_COLOR равен 5
```

Здесь создано хранилище и проведена диспетчеризация действия, добавляющего новый цвет, за которым последовало действие, изменившее его рейтинг. В выводе на консоль показывается, что выполненная диспетчеризация действий фактически изменила состояние.

Изначально в массиве было три цвета. Теперь после добавления образца *x* стало четыре. Начальный рейтинг нового цвета был равен нулю. Диспетчеризация изменила его на 5. Единственный способ изменения данных заключается в диспетчеризации действий в хранилище.

Подписка на хранилища

Хранилища позволяют осуществлять подписку функций-обработчиков, вызываемых после каждого завершения хранилищем диспетчеризации действия. В следующем примере в консоль будет выводиться количество цветов:

```
store.subscribe(() =>
  console.log('color count:', store.getState().colors.length)
)

store.dispatch({
  type: "ADD_COLOR",
  id: "2222e1p5-3abl-0p523-30e4-800118yf2222",
  title: "Party Pink",
  color: "#F142FF",
  timestamp: "Thu Mar 10 2016 01:11:12 GMT-0800 (PST)"
})

store.dispatch({
  type: "ADD_COLOR",
  id: "3315e1p5-3abl-0p523-30e4-800118yf2412",
```

```

    title: "Big Blue",
    color: "#0000FF",
    timestamp: "Thu Mar 10 2016 01:11:12 GMT-0800 (PST)"
  })

store.dispatch({
  type: "RATE_COLOR",
  id: "2222e1p5-3abl-0p523-30e4-800118yf2222",
  rating: 5
})

store.dispatch({
  type: "REMOVE_COLOR",
  id: "3315e1p5-3abl-0p523-30e4-800118yf2412"
})

// Вывод на консоль
// color count: 1
// color count: 2
// color count: 2
// color count: 1

```

Подписка этого слушателя на хранилище приведет к выводу в консоль количества цветов при каждой отправке действия. В предыдущем примере мы видели четыре вывода в консоль: первые два для `ADD_COLOR`, третий для `RATE_COLOR` и четвертый — для `REMOVE_COLOR`.

Принадлежащий хранилищу метод `subscribe` возвращает функцию, которую позже можно использовать для прекращения подписки слушателя:

```

const logState = () => console.log('next state', store.getState())

const unsubscribeLogger = store.subscribe(logState)

// Вызвать по готовности к прекращению подписки слушателя
unsubscribeLogger()

```

Сохранение в localStorage

Используя принадлежащую хранилищу функцию подписки `subscribe`, мы будем отслеживать изменения состояния и сохранять эти изменения в локальном хранилище `localStorage` под ключом `'redux-store'`. При создании хранилища можно проверить, не сохранены ли какие-либо данные под этим ключом, и если да, то загрузить их в качестве исходного состояния. Применяя всего лишь несколько строк кода, мы можем иметь в браузере постоянные данные состояния:

```

const store = createStore(
  combineReducers({ colors, sort }),
  (localStorage['redux-store']) ?
    JSON.parse(localStorage['redux-store']) :
    {}
)

```

```
store.subscribe(() => {
  localStorage['redux-store'] = JSON.stringify(store.getState())
})

console.log('current color count', store.getState().colors.length)
console.log('current state', store.getState())

store.dispatch({
  type: "ADD_COLOR",
  id: uuid.v4(),
  title: "Party Pink",
  color: "#F142FF",
  timestamp: new Date().toString()
})
```

При каждом обновлении этого кода список цветов прибавляет один. Сначала, в вызове функции `createStore`, проверяется наличие ключа `redux-store`. Если он присутствует, то выполняется парсинг JSON. В противном случае возвращается пустой объект. Затем проводится подписка слушателя на хранилище, которая сохраняет состояние, имеющееся в хранилище при каждой диспетчеризации действия. Обновление страницы продолжит добавление того же цвета.

Итак, хранилища в Redux-приложениях содержат данные состояния и управляют ими, и единственный способ их изменения заключается в диспетчеризации действий через хранилище. Хранилище содержит состояние приложения в виде единственного объекта. Управлять изменениями в хранилище можно через преобразователи. Хранилища создаются путем предоставления преобразователя вместе с дополнительными данными для начального состояния. Кроме того, на хранилище можно подписывать слушателей (а позже отменять их подписку), и они станут вызываться при каждом завершении хранилищем диспетчеризации действия.

Создатели действий

Объекты действий представляют собой простые литералы JavaScript. *Создатели* действий являются функциями, которые создают и возвращают эти литералы. Рассмотрим следующие действия:

```
{
  type: "REMOVE_COLOR",
  id: "3315e1p5-3ab1-0p523-30e4-800118yf2412"
}

{
  type: "RATE_COLOR",
  id: "441e0p2-9ab4-0p523-30e4-800118yf2412",
  rating: 5
}
```

Логику, используемую при создании действия, можно упростить, добавив создателей к каждому из этих типов действий:

```
import C from './constants'

export const removeColor = id =>
  ({
    type: C.REMOVE_COLOR,
    id
  })

export const rateColor = (id, rating) =>
  ({
    type: C.RATE_COLOR,
    id,
    rating
  })
```

Теперь, как только понадобится выполнить диспетчеризацию `RATE_COLOR` или `REMOVE_COLOR`, мы можем использовать создатель действия и отправить необходимые данные в виде аргументов функций:

```
store.dispatch( removeColor("3315e1p5-3ab1-0p523-30e4-800118yf2412") )
store.dispatch( rateColor("441e0p2-9ab4-0p523-30e4-800118yf2412", 5) )
```

Создатели упрощают задачу диспетчеризации действий; нам остается лишь вызвать функцию и отправить ей необходимые данные. Создатели способны абстрагировать от подробностей того, как создается действие, что может существенно упростить процесс создания действия. Например, если создается действие под названием `sortBy`, то создатель может принять решение о соответствующих мерах по его реализации:

```
import C from './constants'

export const sortColors = sortedBy =>
  (sortedBy === "rating") ?
    ({
      type: C.SORT_COLORS,
      sortBy: "SORTED_BY_RATING"
    }) :
  (sortedBy === "title") ?
    ({
      type: C.SORT_COLORS,
      sortBy: "SORTED_BY_TITLE"
    }) :
    ({
      type: C.SORT_COLORS,
      sortBy: "SORTED_BY_DATE"
    })
```

Создатель действий `sortColors` проверяет сортировщика `sortedBy` на `"rating"` (сортировка по рейтингу), на `"title"` (сортировка по названию) и может воспользоваться

вариантом по умолчанию. Теперь, когда нужно выполнить диспетчеризацию действия `sortColors`, приходится набирать значительно меньше текста:

```
store.dispatch( sortColors("title") )
```

У создателей действий имеется логика. Кроме того, при создании действия они также могут помочь абстрагироваться от ненужных подробностей. Посмотрим, к примеру, на действие для добавления цвета:

```
{
  type: "ADD_COLOR",
  id: uuid.v4(),
  title: "Party Pink",
  color: "#F142FF",
  timestamp: new Date().toString()
}
```

До сих пор идентификаторы и метки времени генерировались при диспетчеризации действий. Перемещение этой логики в создатель действий приведет к абстрагированию процесса диспетчеризации действий от подробностей:

```
import C from './constants'
import { v4 } from 'uuid'

export const addColor = (title, color) =>
  ({
    type: C.ADD_COLOR,
    id: v4(),
    title,
    color,
    timestamp: new Date().toString()
  })
```

Создатель действий `addColor` сгенерирует уникальный идентификатор и предоставит метку времени. Теперь создавать новые цвета намного проще: уникальный ID будет предоставляться при создании переменной, значение которой можно увеличивать на единицу, а метка времени будет устанавливаться автоматически с использованием настоящего времени на стороне клиента:

```
store.dispatch( addColor("#F142FF", "Party Pink") )
```

Создатели действий хороши тем, что предоставляют место для инкапсуляции всей логики, необходимой для успешного создания действия. Создатель действий `addColor` берет на себя все связанное с добавлением новых цветов, включая предоставление уникальных идентификаторов и установку на действие метки времени. Все перечисленное находится в одном месте, что существенно упрощает отладку приложения.

В создатель действий нужно помещать всю логику обмена данными с серверными API. Используя создатель, можно задействовать логику асинхронного обмена

данными, такую как запрос данных или вызов функций API. Эти вопросы мы обсудим в главе 12, когда представим работу с сервером.

compose

Redux также поставляется с функцией `compose`, которой можно воспользоваться для составления нескольких функций в одну. Это похоже на составление комбинации функций в главе 3, но приводит к более надежному результату. Кроме того, функции составляются не слева направо, а наоборот.

Если нужен просто список цветов с запятыми в качестве разделителей, то можно воспользоваться следующей весьма необычной по виду строкой кода:

```
console.log(store.getState().colors.map(c=>c.title).join(", "))
```

Более функциональным был бы подход, при котором все разбивалось бы на меньшие функции и составлялось в единую функцию:

```
import { compose } from 'redux'

const print = compose(
  list => console.log(list),
  titles => titles.join(", "),
  map => map(c=>c.title),
  colors => colors.map.bind(colors),
  state => state.colors
)

print(store.getState())
```

Функция `compose` получает свои функции в виде аргументов и первой вызывает самую правую. Сначала из состояния берутся цвета, затем возвращается связанная функция отображения `map`, за ней следует массив названий цветов, элементы которого объединяются в список с запятыми в качестве разделителей, и, наконец, этот список выводится на консоль.

Функции промежуточного звена

Если когда-либо вам приходилось использовать такие серверные среды, как Express, Sinatra, Django, KOA или ASP.NET, то вы, наверное, уже знакомы с концепцией *связующего (или промежуточного) кода* (middleware). (Если же нет, то связующий код служит в качестве соединителя между различными уровнями или различными частями программного средства.)

В Redux тоже есть связующий код. Он работает в конвейере диспетчеризации, действующем в хранилище. Связующий код в Redux состоит из серии функций, выполняемых поочередно в процессе диспетчеризации действия (рис. 8.7).

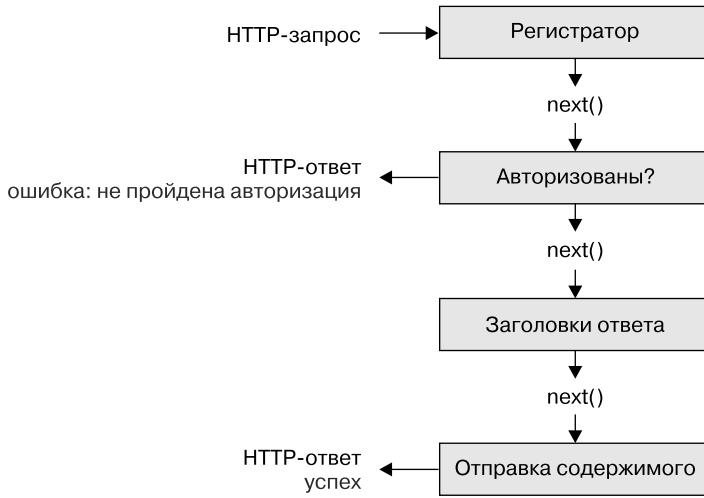


Рис. 8.7. Конвейер связующего кода HTTP-запроса

Эти функции высшего порядка позволяют вставлять функциональные средства до или после диспетчеризации действий и обновления состояния. Все функции связующего кода выполняются последовательно (рис. 8.8).

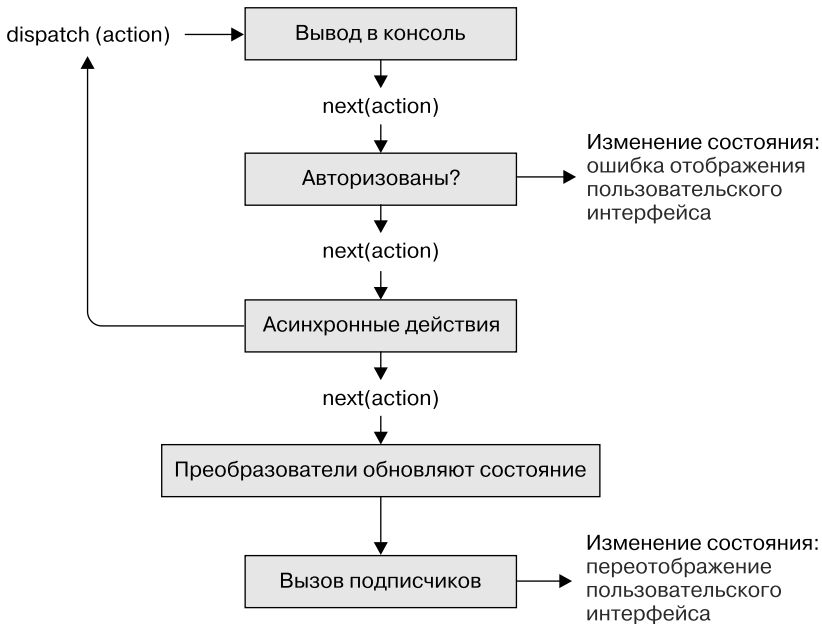


Рис. 8.8. Функции связующего кода выполняются последовательно

Каждая часть связующего кода является функцией, имеющей доступ к действию, функцией `dispatch`, и функцией `next`, которая вызывает обновление. Перед ее вызовом можно изменить действие. После вызова в состояние уже будут внесены изменения.

Применение связующего кода к хранилищу. В этом разделе мы намереваемся создать фабрику под названием `storeFactory`. *Фабрикой* называется функция, которая управляет процессом создания хранилищ. В данном случае фабрика создаст хранилище, которое имеет связующий код для регистрации и сохранения данных. Фабрика `storeFactory` будет в виде единого файла, содержащего одну функцию, выполняющую группировку всего необходимого для создания хранилища. Как только оно понадобится, может быть вызвана эта функция:

```
const store = storeFactory(initialData)
```

При создании хранилища вводятся две части связующего кода: *регистратор* (`logger`) и *сохранитель* (`saver`) (пример 8.9). Данные вместо метода `store` сохраняются в `localStorage` с помощью связующего кода.

Пример 8.9. `storeFactory: ./store/index.js`

```
import { createStore,
        combineReducers,
        applyMiddleware } from 'redux'
import { colors, sort } from './reducers'
import stateData from './initialState'

const logger = store => next => action => {
  let result
  console.groupCollapsed("dispatching", action.type)
  console.log('prev state', store.getState())
  console.log('action', action)
  result = next(action)
  console.log('next state', store.getState())
  console.groupEnd()
}

const saver = store => next => action => {
  let result = next(action)
  localStorage['redux-store'] = JSON.stringify(store.getState())
  return result
}

const storeFactory = (initialState=stateData) =>
  applyMiddleware(logger, saver)(createStore)(
    combineReducers({colors, sort}),
    (localStorage['redux-store'] ?
      JSON.parse(localStorage['redux-store']) :
      stateData
    )
  )

export default storeFactory
```

И `logger`, и `saver` являются функциями промежуточного звена. В Redux связующий код определяется в качестве функций высшего порядка, то есть функций, возвращающих функцию, которая возвращает функцию. Последняя возвращенная функция вызывается при каждой диспетчеризации действия. В момент вызова появляется доступ к действию, к хранилищу и к функции для отправки действия к следующему связующему коду.

Вместо непосредственного экспортирования хранилища мы экспортируем функцию, то есть фабрику, которая может использоваться для создания хранилищ. После вызова она создаст и возвратит хранилище с включенными в него функциями регистрации и сохранения.

В регистраторе, прежде чем действие пройдет диспетчеризацию, мы открываем новую консольную группу и регистрируем текущее состояние и текущее действие. За счет вызова функции `next` действие передается по конвейеру к следующей части связующего кода и попадает в конечном счете к преобразователям. К этому моменту состояние уже будет обновлено, так что мы регистрируем измененное состояние и завершаем работу консольной группы.

В сохранителе функция `next` вызывается с действием, что приводит к изменению состояния. Затем новое состояние сохраняется в `localStorage` и возвращается результат (см. пример 8.9).

В примере 8.10 с помощью `storeFactory` создается экземпляр хранилища. Поскольку никакие аргументы этому хранилищу посланы не были, исходное состояние поступит из данных состояния.

Пример 8.10. Создание хранилища с помощью фабрики

```
import storeFactory from "./store"

const store = storeFactory(true)

store.dispatch( addColor("#FFFFFF", "Bright White") )
store.dispatch( addColor("#00FF00", "Lawn") )
store.dispatch( addColor("#0000FF", "Big Blue") )
```

Каждое действие, проходящее диспетчеризацию из этого хранилища, станет добавлять к консоли новую группу регистрационных записей, а новое состояние будет сохранено в `localStorage`.

В данной главе мы рассмотрели все основные характерные особенности Redux: состояние, действия, преобразователи, хранилища, создатели хранилищ и функции промежуточного звена, образующие связующий код. Все состояние приложения мы обрабатывали с помощью Redux, и теперь его можно подключить к нашему пользовательскому интерфейсу.

В следующей главе мы обсудим фреймворк `react-redux` — инструмент, позволяющий эффективно соединять наше хранилище Redux с пользовательским интерфейсом React.

9

React Redux

Из главы 6 мы узнали, как создавать компоненты React. С помощью доступной в React системы управления состоянием мы разработали органайзер цветов. В предыдущей главе обсудили, как использовать Redux для управления данными состояния нашего приложения, и завершили создание хранилища для органайзера, готового к диспетчеризации действий. В этой главе мы собираемся объединить пользовательский интерфейс, созданный в главе 6, с хранилищем, созданным в предыдущей главе.

Приложение, разработанное в главе 6, хранит состояние в одном объекте в одном и том же месте — в компоненте App.

```
export default class App extends Component {
  constructor(props) {
    super(props)
    this.state = {
      colors: [
        {
          "id": "8658c1d0-9eda-4a90-95e1-8001e8eb6036",
          "title": "Ocean Blue",
          "color": "#0070ff",
          "rating": 3
        },
        {
          "id": "f9005b4e-975e-433d-a646-79df172e1dbb",
          "title": "Tomato",
          "color": "#d10012",
          "rating": 2
        },
        {
          "id": "58d9caee-6ea6-4d7b-9984-65b145031979",
          "title": "Lawn",
          "color": "#67bf4f",
          "rating": 1
        }
      ]
    }
  }
}
```

```

    },
    {
      "id": "a5685c39-6bdc-4727-9188-6c9a00bf7f95",
      "title": "Party Pink",
      "color": "#ff00f7",
      "rating": 5
    }
  ]
}
this.addColor = this.addColor.bind(this)
this.rateColor = this.rateColor.bind(this)
this.removeColor = this.removeColor.bind(this)
}

addColor(title, color) {
  ...
}

rateColor(id, rating) {
  ...
}

removeColor(id) {
  ...
}

render() {
  const { addColor, rateColor, removeColor } = this
  const { colors } = this.state
  return (
    <div className="app">
      <AddColorForm onNewColor={addColor} />
      <ColorList colors={colors}
                onRate={rateColor}
                onRemove={removeColor} />
    </div>
  )
}
}

```

Именно в компоненте `App` и сохраняется состояние. Оно передается вниз дочерним компонентам в виде свойств. В частности, цвета передаются в виде свойства из состояния компонента `App` компоненту `ColorList`. В случае ошибки данные передаются обратно вверх по дереву компоненту `App` через свойства функции обратного вызова (рис. 9.1).

Процесс передачи данных все время вниз и вверх по дереву создает сложности, для преодоления которых созданы такие библиотеки, как `Redux`. Вместо передачи данных вверх по дереву через двустороннюю привязку функций можно выполнять диспетчеризацию действий непосредственно из дочерних компонентов для обновления состояния приложения.

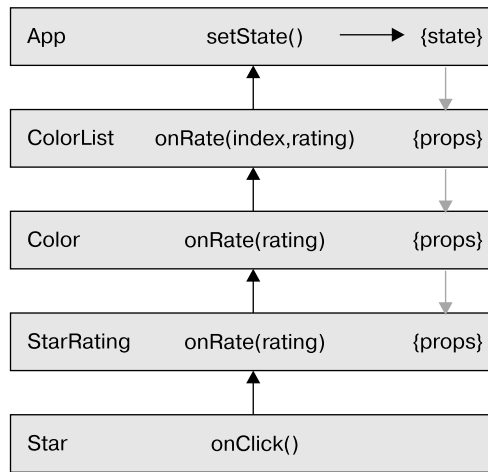


Рис. 9.1. Поток данных по дереву компонентов

В этой главе мы рассмотрим различные способы применения хранилища Redux. Сначала обсудим возможный способ использования хранилища без применения дополнительных сред. Затем исследуем ReactRedux, фреймворк, который можно задействовать для объединения хранилища Redux с компонентами React.

Явная передача хранилища

Первым и наиболее логичным шагом в применении хранилища в пользовательском интерфейсе станет передача его явным образом вниз по дереву компонентов в виде свойства. Это весьма простой и вполне работоспособный подход для небольших приложений, имеющих всего несколько вложенных компонентов.

Посмотрим, как можно применить хранилище в органайзере цветов. В коде файла `./index.js` отобразим компонент `App`, которому будет передано хранилище:

```
import React from 'react'
import ReactDOM from 'react-dom'
import App from './components/App'
import storeFactory from './store'

const store = storeFactory()

const render = () =>
  ReactDOM.render(
    <App store={store}/>,
    document.getElementById('react-container')
  )

store.subscribe(render)
render()
```

Это файл `./index.js`; в нем с помощью фабрики `storeFactory` создается хранилище, и компонент `App` отображается в документе. При отображении компоненту `App` передается хранилище в виде свойства. При каждом внесении изменений в хранилище будет вызываться функция `render`, которая рациональным образом обновит пользовательский интерфейс с новыми данными состояния.

После передачи хранилища компоненту `App` необходимо продолжить его передачу вниз, тем дочерним компонентам, которые в нем нуждаются:

```
import AddColorForm from './AddColorForm'
import SortMenu from './SortMenu'
import ColorList from './ColorList'

const App = ({ store }) =>
  <div className="app">
    <SortMenu store={store} />
    <AddColorForm store={store} />
    <ColorList store={store} />
  </div>

export default App
```

Компонент `App` — корневой. Он получает хранилище из свойств и явным образом передает его вниз, своим дочерним компонентам. Хранилище передается компонентам `SortMenu`, `AddColorForm` и `ColorList` в виде свойства.

После передачи из компонента `App` хранилище можно использовать внутри дочерних компонентов. Стоит напомнить, что состояние хранилища можно прочитать с помощью метода `store.getState`, а диспетчеризацию действий к хранилищу выполнять с применением метода `store.dispatch`.

Из компонента `AddColorForm` можно воспользоваться хранилищем для диспетчеризации действий `ADD_COLOR`. Когда пользователь отправляет данные формы, цвет и его название собираются из ссылок, и эти данные служат для создания и диспетчеризации нового действия `ADD_COLOR`:

```
import { PropTypes, Component } from 'react'
import { addColor } from '../actions'

const AddColorForm = ({store}) => {

  let _title, _color
  const submit = e => {
    e.preventDefault()
    store.dispatch( addColor(_title.value, _color.value) )
    _title.value = ''
    _color.value = '#000000'
    _title.focus()
  }

  return (
    <form className="add-color" onSubmit={submit}>
```



```

      <input ref={input => _title = input}
        type="text"
        placeholder="color title..." required/>
      <input ref={input => _color = input}
        type="color" required/>
      <button>ADD</button>
    </form>
  )
}

AddColorForm.propTypes = {
  store: PropTypes.object
}

export default AddColorForm

```

В коде этого компонента импортируется необходимый создатель действия `addColor`. Когда пользователь отправляет данные формы, новое действие `ADD_COLOR` направляется непосредственно в хранилище с помощью `addColor`.

Компонент `ColorList` может воспользоваться принадлежащим хранилищу методом `getState`, чтобы получить исходные цвета и провести их соответствующую сортировку. Он также способен напрямую выполнять диспетчеризацию действий `RATE_COLOR` и `REMOVE_COLOR` по мере их возникновения:

```

import { PropTypes } from 'react'
import Color from './Color'
import { rateColor, removeColor } from '../actions'
import { sortFunction } from '../lib/array-helpers'

const ColorList = ({ store }) => {
  const { colors, sort } = store.getState()
  const sortedColors = [...colors].sort(sortFunction(sort))
  return (
    <div className="color-list">
      {(colors.length === 0) ?
        <p>No Colors Listed. (Add a Color)</p> :
        sortedColors.map(color =>
          <Color key={color.id}
            {...color}
            onRate={(rating) =>
              store.dispatch(
                rateColor(color.id, rating)
              )
            }
            onRemove={() =>
              store.dispatch(
                removeColor(color.id)
              )
            }
          </>
        )
      }
    </div>
  )
}

```

```
ColorList.propTypes = {  
  store: PropTypes.object  
}  
  
export default ColorList
```

Хранилище прошло весь путь вниз по дереву компонентов до компонента `ColorList`. Он напрямую взаимодействует с хранилищем. При оценке или удалении цветов соответствующие действия направляются хранилищу.

Кроме того, хранилище используется для получения исходных цветов. Они дублируются и сортируются в соответствии со значением принадлежащего хранилищу свойства `sort` и сохраняются как `sortedColors`. Затем `sortedColors` применяется для создания пользовательского интерфейса.

Описанный подход хорош для небольшого дерева компонентов, как у этого органайзера. Негативная сторона его использования такова: нам приходится непосредственно передавать хранилище дочерним компонентам, что несколько увеличивает объем кода и приносит больше хлопот по сравнению с другими подходами. Кроме того, именно это хранилище требуется компонентам `SortMenu`, `AddColorForm` и `ColorList`. При таких условиях будет сложно применять их повторно в других приложениях.

В следующих двух разделах мы рассмотрим другие способы предоставления хранилища тем компонентам, которые в нем нуждаются.

Передача хранилища через контекст

В предыдущем разделе мы создали хранилище, которое затем прошло по всему пути вниз по дереву компонентов из компонента `ColorList`. При таком подходе требуется проход хранилища через каждый компонент, встречающийся на пути между `App` и `ColorList`.

Предположим, что имеется груз, который нужно перевезти из Вашингтона, округ Колумбия, в Сан-Франциско, Калифорния. Можно воспользоваться поездом, но для этого придется проложить путь через как минимум девять штатов, чтобы наш груз добрался до конечной точки. Это похоже на явную передачу хранилища вниз по дереву компонентов от корневого компонента к компонентам-листам. Приходится «прокладывать путь» через каждый компонент, встречающийся на пути между пунктом отправки и пунктом доставки. Если использование поезда похоже на явную передачу хранилища с помощью свойств, то его явная передача через контекст сродни применению реактивного самолета. При перелете из округа Колумбия в Сан-Франциско он минует как минимум девять штатов, и никакие пути прокладывать не нужно.

Точно так же можно воспользоваться преимуществом функционального свойства `React`, называемого *контекстом*, которое позволяет передавать переменные

компонентам без обязательной явной передачи их вниз по дереву в виде свойств¹. Доступ к этим контекстным переменным могут получить любые дочерние компоненты.

Если бы нам понадобилось в органайзере цветов передавать хранилище с помощью контекста, первым шагом была бы реструктуризация компонента `App` для хранения контекста. Компоненту `App` придется также отслеживать хранилище, чтобы можно было запускать обновление пользовательского интерфейса при каждом изменении состояния:

```
import { PropTypes, Component } from 'react'
import SortMenu from './SortMenu'
import ColorList from './ColorList'
import AddColorForm from './AddColorForm'
import { sortFunction } from '../lib/array-helpers'

class App extends Component {

  getChildContext() {
    return {
      store: this.props.store
    }
  }

  componentWillMount() {
    this.unsubscribe = store.subscribe(
      () => this.forceUpdate()
    )
  }

  componentWillUnmount() {
    this.unsubscribe()
  }

  render() {
    const { colors, sort } = store.getState()
    const sortedColors = [...colors].sort(sortFunction(sort))
    return (
      <div className="app">
        <SortMenu />
        <AddColorForm />
        <ColorList colors={sortedColors} />
      </div>
    )
  }
}

App.propTypes = {
  store: PropTypes.object.isRequired
}

App.childContextTypes = {
```

¹ *Abramov D.* Redux: Extracting Container Components. <https://egghead.io/lessons/javascript-redux-passing-the-store-down-implicitly-via-context>, Egghead.io.

```

    store: PropTypes.object.isRequired
  }

```

```
export default App
```

Сначала добавление контекста к компоненту потребует использования функции жизненного цикла `getChildContext`. Она будет возвращать объект, определяющий контекст. В данном случае хранилище добавляется к контексту, к которому можно получить доступ с помощью свойств.

Затем нужно в экземпляре компонента указать `childContextTypes` и определить ваш объект контекста. Это аналогично добавлению к экземпляру компонента `propTypes` или `defaultProps`. Но, чтобы контекст заработал, без данного шага не обойтись.

Теперь любой дочерний по отношению к `App` компонент будет иметь доступ к хранилищу через контекст. Дочерние компоненты смогут напрямую совершать вызовы `store.getState` и `store.dispatch`. Завершающим шагом будет подписка на хранилище и обновление дерева компонентов при каждом обновлении хранилищем состояния приложения. Здесь пригодятся функции жизненного цикла установки (см. одноименный подраздел в главе 7). В функции `componentWillMount` можно открыть подписку на хранилище и воспользоваться выражением `this.forceUpdate` для запуска жизненного цикла обновления, который заново отобразит пользовательский интерфейс. В функции `componentWillUnmount` можно вызвать функцию отказа от подписки `unsubscribe` и прекратить отслеживание хранилища. Поскольку компонент `App` сам запускает обновления пользовательского интерфейса, больше не нужно подписываться на хранилище из записи файла `./index.js`; отслеживание изменений в хранилище происходит из того же самого компонента, который добавляет хранилище к контексту, то есть из `App`.

Реструктурируем компонент `AddColorForm` для извлечения хранилища и непосредственной диспетчеризации действия `ADD_COLOR`:

```

const AddColorForm = (props, { store }) => {
  let _title, _color

  const submit = e => {
    e.preventDefault()
    store.dispatch(addColor(_title.value, _color.value))
    _title.value = ''
    _color.value = '#000000'
    _title.focus()
  }

  return (
    <form className="add-color" onSubmit={submit}>
      <input ref={input => _title = input}
        type="text"
        placeholder="color title..." required/>
      <input ref={input => _color = input}
        type="color" required/>
    </form>
  )
}

```

```

        <button>ADD</button>
      </form>
    )
  }
}

```

```

AddColorForm.contextTypes = {
  store: PropTypes.object
}

```

Объект контекста передается функциональным компонентам, не имеющим состояния, в виде второго аргумента, следующего после свойств. Для получения хранилища из этого объекта в аргументах можно воспользоваться деструктуризацией объекта. Чтобы получить хранилище, в экземпляре `AddColorForm` нужно определить `contextTypes`. Тем самым React узнает, какие контекстные переменные задействует этот компонент. Без данного шага не обойтись, поскольку хранилище нельзя извлечь из контекста.

Посмотрим на то, как использовать контекст в классе компонента. Компонент `Color` может извлечь хранилище и провести непосредственную диспетчеризацию действий `RATE_COLOR` и `REMOVE_COLOR`:

```

import { PropTypes, Component } from 'react'
import StarRating from './StarRating'
import TimeAgo from './TimeAgo'
import FaTrash from 'react-icons/lib/fa/trash-o'
import { rateColor, removeColor } from '../actions'

class Color extends Component {

  render() {
    const { id, title, color, rating, timestamp } = this.props
    const { store } = this.context
    return (
      <section className="color" style={this.style}>
        <h1 ref="title">{title}</h1>
        <button onClick={() =>
          store.dispatch(
            removeColor(id)
          )
        }>
          <FaTrash />
        </button>
        <div className="color"
          style={{ backgroundColor: color }}>
        </div>
        <TimeAgo timestamp={timestamp} />
        <div>
          <StarRating starsSelected={rating}
            onRate={rating =>
              store.dispatch(
                rateColor(id, rating)
              )
            } />
        </div>
      </section>
    )
  }
}

```

```

        </div>
      </section>
    )
  }
}

Color.contextTypes = {
  store: PropTypes.object
}

Color.propTypes = {
  id: PropTypes.string.isRequired,
  title: PropTypes.string.isRequired,
  color: PropTypes.string.isRequired,
  rating: PropTypes.number
}

Color.defaultProps = {
  rating: 0
}

export default Color

```

Теперь `ColorList` является классом компонента и может обращаться к контексту через выражение `this.context`. Сейчас образцы цвета считываются непосредственно из хранилища через `store.getState`. Здесь применяются те же правила, что и для функциональных компонентов, не имеющих состояния. В экземпляре должно быть определение `contextTypes`.

Извлечение хранилища из контекста — хороший способ сократить объем повторяющегося кода, но нет нужды применять его для каждого приложения. Дэн Абрамов (Dan Abramov), создатель Redux, даже намекнул, что этим моделям не нужно следовать в обязательном порядке.

«Зачастую отделение контейнера от презентационных компонентов представляется вполне рациональной идеей, но не следует воспринимать это как догму. Используйте данную модель только в том случае, если она действительно упрощает ваш исходный код»¹.

Сравнение презентационных и контейнерных КОМПОНЕНТОВ

В предыдущем примере компонент `Color` извлекал хранилище через контекст и использовал его для непосредственной диспетчеризации `RATE_COLOR` и `REMOVE_COLOR`. Перед этим компонент `ColorList` извлекал хранилище через контекст, чтобы прочесть текущий список цветов из состояния. В обоих примерах эти компоненты

¹ Документация React: Context. <https://facebook.github.io/react/docs/context.html>.

занимались повторным отображением пользовательского интерфейса с помощью непосредственного взаимодействия с хранилищем Redux. Мы можем усовершенствовать архитектуру нашего приложения, отвязав хранилище от компонентов, отображающих UI¹.

Презентационные компоненты занимаются исключительно отображением элементов пользовательского интерфейса². Они не имеют тесной связи с любыми данными архитектуры. Вместо этого получают данные в виде свойств и отправляют сведения своему родительскому компоненту через свойства функции обратного вызова. Презентационные компоненты занимаются исключительно пользовательским интерфейсом и могут повторно использоваться в разных приложениях, содержащих разные данные. Таковым, за исключением компонента App, является каждый компонент, созданный нами в главе 6.

Контейнерные компоненты подключают презентационные компоненты к данным. В нашем случае контейнерные компоненты будут извлекать состояние через контекст и управлять любыми операциями по взаимодействию с хранилищем. Они выводят на экран презентационные компоненты путем отображения свойств на состояние и свойств функций обратного вызова на принадлежащий хранилищу метод `dispatch`. Контейнерные компоненты не имеют никакого отношения к элементам пользовательского интерфейса и используются только для подключения презентационных компонентов к данным.

У такой архитектуры имеется множество преимуществ. Презентационные компоненты можно задействовать многократно. Они легко заменяются и легко тестируются. Для создания пользовательского интерфейса из них можно составлять композиции. Эти компоненты способны повторно использоваться разными браузерными приложениями, в которых могут применяться различные библиотеки данных.

Контейнерные компоненты пользовательским интерфейсом не занимаются. Они нацелены в основном на подключение презентационных компонентов к архитектуре данных. Контейнерные компоненты могут повторно использоваться на аппаратных платформах для подключения присущих им презентационных компонентов к данным.

Примерами презентационных компонентов могут послужить созданные нами в главе 6 компоненты `AddColorForm`, `ColorList`, `Color`, `StarRating` и `Star`. Они получают данные через свойства и, когда происходят события, задействуют свойства функции обратного вызова. С презентационными компонентами мы уже

¹ Документация Redux: Presentational and Container Components. <http://redux.js.org/docs/basics/UsageWithReact.html>.

² Abramov D. Presentational and Container Components. https://medium.com/@dan_abramov/smart-and-dumb-components-7ca2f9a7c7d0.

познакомились и теперь посмотрим, как ими можно воспользоваться для создания контейнерных компонентов.

Компонент `App` остался в основном в прежнем виде. В нем просто было определено хранилище в контексте, чтобы его могли извлекать дочерние элементы. Но вместо отображения компонентов `SortMenu`, `AddColorForm` и `ColorList` он будет отображать предназначенные для них контейнеры. Контейнер `Menu` будет подключен к `SortMenu`, `NewColor` — к `AddColorForm`, а `Colors` — к `ColorList`:

```
render() {
  return (
    <div className="app">
      <Menu />
      <NewColor />
      <Colors />
    </div>
  )
}
```

Когда понадобится подключить презентационный компонент к неким данным, этот компонент можно будет заключить в контейнер, управляющий свойствами и подключающий их к данным. Все контейнеры: `NewColor`, `Menu` и `Colors` — могут быть определены в одном и том же файле:

```
import { PropTypes } from 'react'
import AddColorForm from './ui/AddColorForm'
import SortMenu from './ui/SortMenu'
import ColorList from './ui/ColorList'
import { addColor,
  sortColors,
  rateColor,
  removeColor } from '../actions'
import { sortFunction } from '../lib/array-helpers'

export const NewColor = (props, { store }) =>
  <AddColorForm onNewColor={(title, color) =>
    store.dispatch(addColor(title, color))
  } />

NewColor.contextTypes = {
  store: PropTypes.object
}

export const Menu = (props, { store }) =>
  <SortMenu sort={store.getState().sort}
    onSelect={sortBy =>
      store.dispatch(sortColors(sortBy))
    } />

Menu.contextTypes = {
  store: PropTypes.object
}
```



```
export const Colors = (props, { store }) => {
  const { colors, sort } = store.getState()
  const sortedColors = [...colors].sort(sortFunction(sort))
  return (
    <ColorList colors={sortedColors}
      onRemove={id =>
        store.dispatch( removeColor(id) )
      }
      onRate={(id, rating) =>
        store.dispatch( rateColor(id, rating) )
      }/>
  )
}

Colors.contextTypes = {
  store: PropTypes.object
}
```

Контейнер `NewColor` не занимается отображением пользовательского интерфейса. Вместо этого он отображает компонент `AddColorForm` и обрабатывает события `onNewColor`, выдаваемые этим компонентом. Этот контейнерный компонент извлекает хранилище из контекста и использует его для диспетчеризации действий `ADD_COLOR`. Он *содержит* компонент `AddColorForm` и подключает его к хранилищу `Redux`.

Контейнер `Menu` отображает компонент `SortMenu`. Он передает из состояния хранилища текущее свойство `sort` и выполняет диспетчеризацию действий `sort`, когда пользователь выбирает другой пункт меню.

Контейнер `Colors` извлекает хранилище через контекст и отображает компонент `ColorList` с цветами из текущего состояния хранилища. Он также обрабатывает события `onRate` и `onRemove`, выдаваемые компонентом `ColorList`. При выдаче этих событий контейнер `Colors` выполняет диспетчеризацию соответствующих действий.

Все функциональные средства `Redux` подключены здесь, в данном файле. Обратите внимание: все создатели действий были импортированы и использованы в одном месте. Это единственный файл, в котором вызываются методы `store.getState` или `store.dispatch`.

Такой подход, отделяющий компоненты пользовательского интерфейса от контейнеров, подключающих их к данным, в целом неплох. Но для небольших проектов, подтверждений каких-либо концепций или прототипов он может быть излишне сложен.

В следующем разделе мы представим новую библиотеку `React Redux`. Она может использоваться для быстрого добавления хранилища `Redux` к контексту и создания контейнерных компонентов.

Провайдер React Redux

React Redux — библиотека, содержащая инструменты, помогающие упростить подразумеваемую передачу хранилища через контекст. Эта библиотека также предоставлена Дэном Абрамовым (Dan Abramov), создателем Redux. Сам Redux можно использовать и без нее. Но применение React Redux упрощает ваш код и может помочь ускорить создание приложений.

Чтобы воспользоваться библиотекой React Redux, ее сначала нужно установить. Сделать это можно с помощью npm (<https://www.npmjs.com/package/react-redux>):

```
npm install react-redux -save
```

Библиотека `react-redux` предоставляет компонент *провайдер*, служащий для настройки хранилища в контексте. В провайдер можно заключить любой элемент React, и тогда его дочерние элементы получают доступ к хранилищу через контекст.

Вместо установки хранилища в компоненте `App` в качестве переменной контекста этот компонент остается без состояния:

```
import { Menu, NewColor, Colors } from './containers'

const App = () =>
  <div className="app">
    <Menu />
    <NewColor />
    <Colors />
  </div>

export default App
```

Провайдер `provider` добавляет хранилище к контексту и обновляет компонент `App` после диспетчеризации действий. Провайдеру нужен один дочерний компонент:

```
import React from 'react'
import { render } from 'react-dom'
import { Provider } from 'react-redux'
import App from './components/App'
import storeFactory from './store'

const store = storeFactory()

render(
  <Provider store={store}>
    <App />
  </Provider>,
  document.getElementById('react-container')
)
```

Провайдер требует передачи хранилища в виде свойства. Он добавляет хранилище к контексту, чтобы оно могло извлекаться любым дочерним компонентом, принадлежащим компоненту `App`. Простое использование провайдера может сэкономить время и упростить код.

После установки провайдера появляется возможность извлекать хранилище через контекст в дочерних контейнерных компонентах. Но библиотека React Redux предоставляет еще один способ быстрого создания контейнерных компонентов, работающих с провайдером, — использование функции `connect`.

Функция connect библиотеки React Redux

Если придерживаться того, чтобы наши компоненты пользовательского интерфейса были чисто презентационными, то при создании контейнерных компонентов можно положиться на React Redux. Эта библиотека поможет создать контейнерные компоненты с помощью отображения текущего состояния хранилища Redux на свойства презентационного компонента. Она также отобразит функции диспетчеризации хранилища на свойства функций обратного вызова. Все это выполняется благодаря функции высшего порядка под названием `connect`.

Используя `connect`, создадим контейнерный компонент цветов `Color`. Контейнер `Color` подключает компонент `ColorList` к хранилищу:

```
import ColorList from './ColorList'

const mapStateToProps = state =>
  ({
    colors: [...state.colors].sort(sortFunction(state.sort))
  })

const mapDispatchToProps = dispatch =>
  ({
    onRemove(id) {
      dispatch(removeColor(id))
    },
    onRate(id, rating) {
      dispatch(rateColor(id, rating))
    }
  })

export const Colors = connect(
  mapStateToProps,
  mapDispatchToProps
)(ColorList)
```

Функция `connect` является функцией высшего порядка, которая возвращает компонент. Здесь нет опечатки или оговорки: это функциональный JavaScript. Функция `connect` ожидает получения двух аргументов: `mapStateToProps` и `mapDispatchToProps`. Оба они являются функциями. Она возвращает функцию, ожидающую презентационный компонент, и заключает ее в контейнер, отправляющий свои данные через свойства.

Первая функция, `mapStateToProps`, вводит состояние в виде аргумента и возвращает объект, который будет отображен на свойства. Свойство `colors` компонента `ColorList` присваивается массиву отсортированных цветов из состояния.

Вторая функция, `mapDispatchToProps`, вводит принадлежащую хранилищу функцию `dispatch` в виде аргумента, который может использоваться при вызове компонентом `ColorList` свойств функции обратного вызова. Когда `ColorList` выдает события `onRate` или `onRemove`, данные, касающиеся цвета, подвергаемого оценке или удалению, применяются и направляются диспетчером в нужное место.

Функция `connect` работает в связке с провайдером. Он добавляет хранилище к контексту, а `connect` создает компоненты, извлекающие хранилище. При использовании `connect` не приходится беспокоиться о контексте.

Все контейнеры могут быть созданы с помощью функции `connect` библиотеки `React Redux` в одном файле:

```
import { connect } from 'react-redux'
import AddColorForm from './ui/AddColorForm'
import SortMenu from './ui/SortMenu'
import ColorList from './ui/ColorList'
import { addColor,
        sortColors,
        rateColor,
        removeColor } from './actions'
import { sortFunction } from './lib/array-helpers'

export const NewColor = connect(
  null,
  dispatch =>
    ({
      onNewColor(title, color) {
        dispatch(addColor(title, color))
      }
    })
)(AddColorForm)

export const Menu = connect(
  state =>
    ({
      sort: state.sort
    }),
  dispatch =>
    ({
      onSelect(sortBy) {
        dispatch(sortColors(sortBy))
      }
    })
)(SortMenu)

export const Colors = connect(
  state =>
    ({
```

```

        colors: [...state.colors].sort(sortFunction(state.sort))
    }},
    dispatch =>
    ({
        onRemove(id) {
            dispatch(removeColor(id))
        },
        onRate(id, rating) {
            dispatch(rateColor(id, rating))
        }
    })
)(ColorList)

```

В данном примере каждый из контейнеров определен с помощью принадлежащей React Redux функции `connect`. Она подключает Redux к чисто презентационным компонентам. Первый аргумент — функция, отображающая переменные состояния на свойства. Второй — функция, выполняющая диспетчеризацию действий при выдаче событий. При необходимости лишь отобразить свойства функции обратного вызова для проведения диспетчеризации в качестве заместителя первого аргумента можно предоставить значение `null`, как это сделано в определении контейнера `NewColor`.

В этой главе мы обсудили различные способы подключения Redux к React. Мы явно передали хранилище вниз по дереву компонентов, к дочерним компонентам, в качестве свойства. Кроме того, неявно передали хранилище через контекст непосредственно тем компонентам, которые нуждаются в его использовании. Отвязали функциональность хранилища от нашей презентации с помощью контейнерных компонентов. И наконец, задействовали `react-redux` для ускоренного подключения хранилища к презентации с применением контекста и контейнерных компонентов.

Теперь у нас есть работоспособное приложение, использующее и React, и Redux. В следующей главе мы рассмотрим способы написания блочных тестов для всех частей этого приложения.

10 Тестирование

Чтобы не отстать от конкурентов, нужно работать быстрее, но притом обеспечивать качество работы. Одним из жизненно необходимых для этого инструментов является *блочное тестирование*, которое, как следует из названия, позволяет проверять каждую часть, или блок, функций нашего приложения¹.

Одно из преимуществ применения функциональной технологии состоит в том, что она подходит для написания тестируемого кода. Чистые функции тестируемы по самой своей природе. Легко поддаются тестированию и неизменяемые элементы. Составление приложений из небольших функций, разработанных для выполнения конкретных задач, приводит к созданию тестируемых функций или блоков кода.

В этом разделе мы покажем технологические приемы, которые можно использовать для блочного тестирования приложений React Redux. Эта глава будет посвящена не только тестированию, но и инструментам, применяемым для развития и улучшения кода и тестов.

ESLint

В большинстве языков программирования код перед запуском нужно откомпилировать. Языки программирования имеют довольно строгие правила стиля оформления кода, и компиляцию нельзя провести, пока код не будет отформатирован соответствующим образом. В JavaScript отсутствует и компилятор, и такие правила. Мы пишем код, скрещиваем пальцы на удачу и запускаем его в браузере, чтобы посмотреть, работает ли он. При этом хорошей новостью для нас является

¹ Быстрое представление о блочном тестировании можно получить в статье Мартина Фаулера (Martin Fowler) Unit Testing по адресу <http://martinfowler.com/bliki/UnitTest.html>.

наличие инструментов, используемых для анализа нашего кода и заставляющих нас придерживаться определенных правил форматирования.

Процесс анализа кода JavaScript называется *указанием* (hinting) или *проверкой соблюдения стандарта оформления кода* (linting). Первичными средствами, применяемыми для анализа JavaScript и предоставления отзыва о форматировании, являются JSHint и JSLint. ESLint (<https://eslint.org/>) — самый новый инструмент проверки кода, поддерживающий усовершенствованный синтаксис JavaScript. Кроме того, ESLint носит модульный характер. Это значит, что можно создавать и совместно использовать дополнительные модули, добавляемые к конфигурации ESLint для расширения его возможностей.

Мы будем работать с дополнительным модулем под названием `eslint-plugin-react` (<https://www.npmjs.com/package/eslint-plugin-react>). Помимо анализа кода JavaScript, этот модуль будет проводить анализ нашего синтаксиса JSX и React.

Выполним глобальную установку `eslint`. Это можно сделать с помощью `npm`:

```
sudo npm install -g eslint
```

Перед использованием ESLint следует определить ряд конфигурационных правил, с которыми нужно согласиться, чтобы впоследствии следовать им. Определения выполняются в конфигурационном файле, находящемся в корневом каталоге нашего проекта. Этот файл может быть в формате JSON или YAML (<http://yaml.org/>). Последний относится к форматам сериализации данных наподобие JSON, но с меньшим объемом синтаксиса, что несколько упрощает его чтение.

ESLint поставляется с инструментом, помогающим настраивать конфигурацию. Существует несколько конфигурационных файлов, разработанных разными компаниями. Этими файлами можно воспользоваться в качестве отправной точки, но можно создать и свой собственный файл.

Конфигурацию ESLint можно создать, запустив команду `eslint --init` и ответив на ряд вопросов о вашем стиле программирования:

```
$ eslint --init
```

```
? How would you like to configure ESLint?
```

```
Answer questions about your style
```

```
? Are you using ECMAScript 6 features? Yes
```

```
? Are you using ES6 modules? Yes
```

```
? Where will your code run? Browser
```

```
? Do you use CommonJS? Yes
```

```
? Do you use JSX? Yes
```

```
? Do you use React? Yes
```

```
? What style of indentation do you use? Spaces
```

```
? What quotes do you use for strings? Single
```

```
? What line endings do you use? Unix
```

```
? Do you require semicolons? No
```

```
? What format do you want your config file to be in? YAML
```

```
Local ESLint installation not found.
Installing eslint, eslint-plugin-react
```

После запуска команды `eslint --init` происходят три действия.

1. ESLint и `eslint-plugin-react` устанавливаются локально в папку `./node_modules`.
2. К файлу `package.json` автоматически добавляются соответствующие зависимости.
3. Создается конфигурационный файл `.eslintrc.yml`, который включается в корневой каталог вашего проекта.

Протестируем ESLint-конфигурацию, создав файл `sample.js`:

```
const gnar = "gnarly";

const info = ({file=__filename, dir=__dirname}) =>
  <p>{dir}: {file}</p>

switch(gnar) {
  default :
    console.log('gnarley')
    break
}
```

К этому файлу есть несколько вопросов, но он не становится причиной возникновения ошибок в браузере. Технически данный код работает вполне удовлетворительно. Запустим для этого файла ESLint и посмотрим, какого отзыва мы от него удостоимся на основе настроенных нами правил:

```
$ ./node_modules/.bin/eslint sample.js
```

```
/Users/alexbanks/Desktop/eslint-learn/sample.js
 1:20 error Strings must use singlequote      quotes
 1:28 error Extra semicolon                  semi
 3:7 error 'info' is defined but never used  no-unused-vars
 3:28 error '__filename' is not defined       no-undef
 3:44 error '__dirname' is not defined        no-undef
 7:5 error Expected indentation of 0 space ch... indent
 8:9 error Expected indentation of 4 space ch... indent
 8:9 error Unexpected console statement      no-console
 9:9 error Expected indentation of 4 space ch... indent
```

```
✓ 9 problems (9 errors, 0 warnings)
```

ESLint проанализировал наш файл-образец и указал на проблемы, основываясь на выбранных нами настройках. Мы видим, что ESLint жалуется на использование в первой строке двойных кавычек и точки с запятой, поскольку в конфигурационном файле `.eslintrc.yml` мы выбрали применение только одинарных кавычек и отказались от точек с запятой. Затем он жалуется на то, что в коде имеется определение функции `info`, которая нигде в нем не задействована, а ESLint этого не терпит.

Кроме того, он жалуется на `__filename` и `__dirname`, поскольку не выполняет автоматическое включение глобальных переменных Node.js. И наконец, ESLint не нравится отступ нашей инструкции `switch` и использование инструкции `console`.

Чтобы ESLint меньше привередничал, мы можем внести в конфигурацию `.eslintrc.yml` ряд изменений:

```
env:
  browser: true
  commonjs: true
  es6: true
extends: 'eslint:recommended'
parserOptions:
  ecmaFeatures:
    experimentalObjectRestSpread: true
    jsx: true
  sourceType: module
plugins:
  - react
rules:
  indent:
    - error
    - 4
    - SwitchCase: 1
  quotes:
    - error
    - single
  semi:
    - error
    - never
  linebreak-style:
    - error
    - unix
  no-console: 0
globals:
  __filename: true
  __dirname: true
```

После открытия `.eslintrc.yml` первое, что вы увидите, — его доступность и легкость прочтения, в чем, собственно, и заключается цель применения формата YAML. Здесь внесены изменения в правила отступа, чтобы под них подпадал отступ, использованный в инструкциях `switch`. Затем добавлено правило `no-console`, не позволяющее ESLint жаловаться на инструкцию `console.log`. И наконец, добавлена пара глобальных переменных, которые ESLint должен игнорировать.

Но, чтобы наш файл следовал нашим же указаниям по стилю оформления кода, в него все же нужно внести пару изменений:

```
const gnar = 'gnarly'

export const info = ({file=__filename, dir=__dirname}) =>
  <p>{dir}: {file}</p>
```

```
switch(gnar) {
  default :
    console.log('gnarly')
    break
}
```

Мы удалили из первой строки точку с запятой и двойные кавычки. Вдобавок экспортировали функцию `info`: теперь ESLint больше не ругается из-за того, что она не используется. В результате всех манипуляций мы получили файл, который проходит тест формата кода.

Команда `eslint .` запустит lint-проверку, то есть проверку соблюдения стандарта оформления кода, во всем каталоге. Для проведения подобной операции вам, скорее всего, захочется, чтобы ESLint проигнорировал некоторые файлы JavaScript. Имена таких файлов или каталогов можно добавить в файл `.eslintignore`:

```
dist/assets/
sample.js
```

Этот файл предписывает ESLint проигнорировать наш новый файл `sample.js`, а также все, что находится в папке `dist/assets`. Если не проигнорировать папку `assets`, ESLint станет анализировать клиентский файл `bundle.js`, по поводу содержания которого возникнет множество возражений.

Добавим к нашему файлу `package.json` сценарий для запуска lint-проверки таким образом:

```
"scripts": {
  "lint": "./node_modules/.bin/eslint ."
}
```

Теперь ESLint будет запускаться каждый раз, когда нам захочется запустить lint-проверку с помощью `npm`, и станет анализировать все файлы нашего проекта за исключением тех, которые мы проигнорировали.

Тестирование Redux

Тестирование играет для Redux важную роль, поскольку эта библиотека работает только с данными и не имеет пользовательского интерфейса. Библиотека Redux поддается тестированию уже по своей природе, так как ее преобразователи являются чистыми функциями, а вставить состояние в хранилище довольно просто. Написание теста преобразователя в первую очередь упрощает понимание сути работы преобразователя. А написание тестов для хранилища и для создателей действий придаст уверенности в том, что уровень ваших клиентских данных функционирует в точном соответствии с предположениями.

В этом разделе мы напишем несколько блочных тестов для компонентов Redux органайзера цветов.

Разработка, основанная на тестировании

Разработка, основанная на тестировании (test-driven development, TDD) — метод работы, а не технология. Здесь речь идет не о том, что у вас просто есть тесты для вашего приложения. Скорее, это метод работы, при котором тесты являются направляющей силой процесса разработки. Чтобы внедрить в практику TDD, нужно придерживаться следующих шагов.

- ❑ *В первую очередь создать тесты.* Наиболее важный шаг. Сначала в тестах объявляется, что именно создается и как должно работать.
- ❑ *Запустить тесты и увидеть свои (красный уровень).* Запуск тестов перед написанием кода и наблюдение за их сбоями.
- ❑ *Создать минимальный объем кода, требуемый для прохождения тестов (зеленый уровень).* Теперь все, что нужно сделать, сводится к прохождению тестов. Особое внимание уделяется прохождению, никакие функциональные свойства, выходящие за рамки тестов, не добавляются.
- ❑ *Реструктурировать как код, так и тесты (золотой уровень).* После успешного выполнения тестов нужно уделить им и коду более пристальное внимание. Старайтесь выразить свой код как можно проще и красивее¹.

Метод TDD отлично подходит для разработки приложения Redux. Обычно, прежде чем создать преобразователь, проще будет сначала разобраться с тем, как он должен работать. Применение TDD позволит создавать и сертифицировать всю структуру данных для компонента или приложения независимо от пользовательского интерфейса.



TDD и обучение

Если TDD для вас в новинку или то же можно сказать о языке, код на котором вы тестируете, то написание теста до создания кода может вызвать определенные трудности. В этом нет ничего неожиданного, и до тех пор, пока вы не освоитесь, написание кода до создания тестов будет в порядке вещей. Попробуйте выполнять работу постепенно, небольшими частями: немного кода, несколько тестов и т. д. Когда вы набьете руку на тестах, будет проще сначала создавать их.

Далее в главе мы создадим тесты для уже существующего кода. С технической точки зрения это не будет практикумом по TDD. Но в следующем подразделе мы сделаем вид, что нашего кода еще не существует, и прочувствуем рабочий процесс TDD.

¹ Дополнительные сведения о такой модели разработки можно получить в публикации Джеффа Маквертера (Jeff McWherter) и Джеймса Бендера (James Bender) Red, Green, Refactor по адресу <https://www.safaribooksonline.com/library/view/professional-test-driven-development/9780470643204/ch004-sec002.html>.

Тестирование преобразователей

Преобразователи (<http://redux.js.org/docs/basics/Reducers.html>) являются чистыми функциями, вычисляющими и возвращающими результаты на основе входящих аргументов. В тесте контролируются вход, текущее состояние и действие. На основе двух последних элементов можно предсказать, каким будет выход преобразователя.

Прежде чем приступить к написанию тестов, нужно установить среду тестирования. Тесты для React и Redux можно создавать с любой средой тестирования JavaScript. Мы воспользуемся Jest, средой тестирования JavaScript, созданной с прицелом на React:

```
sudo npm install -g jest
```

Запуск этой команды приведет к глобальной установке Jest и Jest CLI. Теперь можно применить команду `jest` из любой папки, чтобы выполнить тесты.

Поскольку мы используем усовершенствованный JavaScript и React, нужно будет транспилировать код и тесты перед их запуском. Чтобы получить такую возможность, достаточно установить пакет `babel-jest`:

```
npm install --save-dev babel-jest
```

После этого перед запуском тестов весь ваш код и тесты будут транспилированы с помощью Babel. Чтобы все заработало, понадобится файл `.babelrc`, но в корневом каталоге проекта у нас уже есть один такой файл.



Пакет `create-react-app`

Проекты, инициализированные с помощью `create-react-app`, уже поступают с установленными пакетами `jest` и `babel-jest`. Кроме того, в корневом каталоге проекта создается каталог `__tests__`.

В среде Jest есть две важные функции настройки тестов: `describe` и `it`. Первая используется для создания набора тестов, а вторая — для каждого теста. Обе функции ожидают предоставления имени теста или набора и функции обратного вызова, в которой содержится тест или набор тестов.

Добавим тестовый файл и заглушим наши тесты. Создайте папку `./__tests__/store/reducers`, а в ней новый файл JavaScript `color.test.js`:

```
describe("color Reducer", () => {  
  it("ADD_COLOR success")  
  it("RATE_COLOR success")  
})
```

В этом примере создан набор тестов для преобразователя цвета, в котором заглушено каждое действие, воздействующее на преобразователь. Каждый тест определен с помощью функции `it`. Можно настроить отложенный тест, просто отправив функции `it` один аргумент.

Активизируйте тест командой `jest`. Запустится среда Jest, и вы получите отчет, что были пропущены два отложенных теста:

```
$ jest

Test Suites: 1 skipped, 0 of 1 total
Tests:       2 skipped, 2 total
Snapshots:   0 total
Time:        0.863s
```

Ran all test suites.



Тестовые файлы

Jest запустит все тесты, находящиеся в каталоге `__tests__`, и все файлы JavaScript в вашем проекте, чьи имена оканчиваются на `.test.js`. Некоторые разработчики предпочитают помещать свои тесты непосредственно за тестируемыми файлами, в то время как другие — группировать свои тесты в одной папке.

Итак, пора создать оба указанных теста. Поскольку проверяется преобразователь цвета, мы импортируем конкретно эту функцию. Она выступит для нас *тестируемой системой* (system under test, SUT). Мы импортируем данную функцию, отправим ей действие и проверим результаты.

Применяемый в Jest обнаружитель совпадений возвращается ожидаемой функцией и используется для проверки результатов. Чтобы протестировать преобразователь цвета, мы задействуем обнаружитель совпадений `.toEqual`. Он проверит соответствие получившегося объекта аргументу, отправленному `toEqual`:

```
import C from '.././././src/constants'
import { color } from '.././././src/store/reducers'

describe("color Reducer", () => {

  it("ADD_COLOR success", () => {
    const state = {}
    const action = {
      type: C.ADD_COLOR,
      id: 0,
      title: 'Test Teal',
      color: '#90C3D4',
      timestamp: new Date().toString()
    }
  })
})
```

```
const results = color(state, action)
expect(results)
  .toEqual({
    id: 0,
    title: 'Test Teal',
    color: '#90C3D4',
    timestamp: action.timestamp,
    rating: 0
  })
})

it("RATE_COLOR success", () => {
  const state = {
    id: 0,
    title: 'Test Teal',
    color: '#90C3D4',
    timestamp: 'Sat Mar 12 2016 16:12:09 GMT-0800 (PST)',
    rating: undefined
  }
  const action = {
    type: C.RATE_COLOR,
    id: 0,
    rating: 3
  }
  const results = color(state, action)
  expect(results)
    .toEqual({
      id: 0,
      title: 'Test Teal',
      color: '#90C3D4',
      timestamp: 'Sat Mar 12 2016 16:12:09 GMT-0800 (PST)',
      rating: 3
    })
  })
})
```

Для тестирования преобразователя необходимо иметь состояние и пример действия. Мы получим результат, запустив нашу SUT (то есть функцию работы с цветом) с этими испытуемыми объектами. И наконец, мы проверим результат, чтобы убедиться в возвращении соответствующего состояния, с помощью обнаружителя `.toEqual`.

Чтобы протестировать действие `ADD_COLOR`, исходное состояние особой роли не играет. Но когда преобразователю цвета отправляется это действие, он должен вернуть новый объект цвета.

Чтобы протестировать действие `RATE_COLOR`, для предполагаемого состояния будет предоставляться исходный объект цвета с рейтингом `0`. Отправка этого объекта состояния наряду с указанным действием должны привести к созданию объекта цвета, имеющего новый рейтинг.

Если после написания тестов сделать вид, что у нас еще нет кода для преобразователя цвета, то нужно заглушить данную функцию. Сделать это можно путем

добавления функции `color` к нашему файлу `/src/store/reducers.js`, что позволит нашим тестам найти пустой преобразователь и выполнить его импорт:

```
import C from '../constants'

export const color = (state={}, action=) => {
  return state
}
```



Зачем сначала заглушать преобразователь?

Не имея SUT, нам нужно получить ошибку прохождения теста:

```
TypeError: (0 , _reducers.color) is not a function
```

Эта ошибка выдается в случае, если тестируемая функция, `color`, не определена. Простое добавление определения для функции, которую нужно протестировать, повлечет выдачу более детальной информации о том, что тест не пройден.

Запустим тесты и посмотрим, как они не будут пройдены. Среда Jest предоставит подробности по каждому сбою, включая трассировку стека:

\$ jest

```
FAIL __tests__/store/reducers/color.test.js
  ● color Reducer > ADD_COLOR success

    expect(received).toEqual(expected)
    Expected value to equal:
      {"color": "#90C3D4", "id": 0, "rating": 0, "timestamp":
        "Mon Mar 13 2017 12:29:12 GMT-0700 (PDT)", "title": "Test Teal"}

    Received:
      {}

    Difference:

    - Expected
    + Received

    @@ -1,7 +1,1 @@
    -Object {
    -  "color": "#90C3D4",
    -  "id": 0,
    -  "rating": 0,
    -  "timestamp": "Mon Mar 13 2017 12:29:12 GMT-0700 (PDT)",
    -  "title": "Test Teal",
    -}

    +Object {}
```

```

    at Object.<anonymous> (__tests__/store/reducers/color.test.js:19:9)
    at process._tickCallback (internal/process/next_tick.js:103:7)

• color Reducer › RATE_COLOR success

expect(received).toEqual(expected)

Expected value to equal:
  {"color": "#90C3D4", "id": 0, "rating": 3, "timestamp":
  "Sat Mar 12 2016 16:12:09 GMT-0800 (PST)", "title": "Test Teal"}

Received:
  {"color": "#90C3D4", "id": 0, "rating": undefined, "timestamp":
  "Sat Mar 12 2016 16:12:09 GMT-0800 (PST)", "title": "Test Teal"}

Difference:

- Expected
+ Received

@@ -1,7 +1,7 @@
Object {
  "color": "#90C3D4",
  "id": 0,
- "rating": 3,
+ "rating": undefined,
  "timestamp": "Sat Mar 12 2016 16:12:09 GMT-0800 (PST)",
  "title": "Test Teal",
}
    at Object.<anonymous> (__tests__/store/reducers/color.test.js:44:9)
    at process._tickCallback (internal/process/next_tick.js:103:7)

color Reducer

  ✓ ADD_COLOR success (8ms)
  ✓ RATE_COLOR success (1ms)

Test Suites: 1 failed, 1 total
Tests:      2 failed, 2 total
Snapshots: 0 total
Time:       0.861s, estimated 1s
Ran all test suites.

```

Мы потратили время на написание и запуск тестов, чтобы посмотреть, как они не будут пройдены, и убедиться в их работе в соответствии с предназначением. Полученные в результате сведения о сбоях представляют собой наш перечень нерешенных задач. Нам нужно добиться выполнения обоих тестов.

Сейчас следует открыть файл `/src/store/reducers.js` и написать минимально необходимый код, позволяющий пройти тесты:

```

import C from '../constants'

export const color = (state={}, action=) => {

```



```

switch (action.type) {
  case C.ADD_COLOR:
    return {
      id: action.id,
      title: action.title,
      color: action.color,
      timestamp: action.timestamp,
      rating: 0
    }
  case C.RATE_COLOR:
    state.rating = action.rating
    return state
  default :
    return state
}
}

```

При следующем запуске команды `jest` наши тесты должны быть пройдены:

```
$ jest
```

```
PASS __tests__/store/reducers/color.test.js
color Reducer
```

```

  ✓ ADD_COLOR success (4ms)
  ✓ RATE_COLOR success

```

```

Test Suites: 1 passed, 1 total
Tests:       2 passed, 2 total
Snapshots:  0 total
Time:        0.513s, estimated 1s

```

```
Ran all test suites.
```

Тесты пройдены, но работа на этом не заканчивается. Настало время реструктуризировать оба теста и код. Посмотрите на `case`-вариант действия `RATE_COLOR` в преобразователе:

```

case 'RATE_COLOR':
  state.rating = action.rating
  return state

```

Если приглядеться, в этом коде что-то не то. Состояние предполагается неизменяемым, а здесь оно явно изменяется путем смены значения `rating` в объекте состояния. И все же код проходит наши тесты, поскольку мы не убеждаемся в неизменности нашего объекта состояния.

Обеспечить сохранность неизменяемости наших объектов состояния и действия может модуль `deep-freeze` (<https://github.com/substack/deep-freeze>), предохраняющий их от изменения:

```
npm install deep-freeze --save-dev
```

При вызове преобразователя цвета с помощью данного модуля мы проведем глубокую заморозку и объекта состояния, и объекта действия. Оба они должны быть

неизменяемыми, и их глубокая заморозка приведет к выдаче ошибки, если какой-либо код попытается внести в них изменения:

```
import C from '.././../src/constants'
import { color } from '.././../src/store/reducers'
import deepFreeze from 'deep-freeze'

describe("color Reducer", () => {

  it("ADD_COLOR success", () => {
    const state = {}
    const action = {
      type: C.ADD_COLOR,
      id: 0,
      title: 'Test Teal',
      color: '#90C3D4',
      timestamp: new Date().toString()
    }
    deepFreeze(state)
    deepFreeze(action)
    expect(color(state, action))
      .toEqual({
        id: 0,
        title: 'Test Teal',
        color: '#90C3D4',
        timestamp: action.timestamp,
        rating: 0
      })
  })

  it("RATE_COLOR success", () => {
    const state = {
      id: 0,
      title: 'Test Teal',
      color: '#90C3D4',
      timestamp: 'Sat Mar 12 2016 16:12:09 GMT-0800 (PST)',
      rating: undefined
    }
    const action = {
      type: C.RATE_COLOR,
      id: 0,
      rating: 3
    }
    deepFreeze(state)
    deepFreeze(action)
    expect(color(state, action))
      .toEqual({
        id: 0,
        title: 'Test Teal',
        color: '#90C3D4',
        timestamp: 'Sat Mar 12 2016 16:12:09 GMT-0800 (PST)',
        rating: 3
      })
  })
})
})
```

Теперь можно запустить модифицированный тест текущего преобразователя цвета и пронаблюдать, как он не будет выполнен, поскольку функция работы с цветом изменяет получаемое состояние:

```
$ jest
```

```
FAIL __tests__/store/reducers/color.test.js
  ● color Reducer › RATE_COLOR success

    TypeError: Cannot assign to read only property 'rating' of object '#<Object>'
      at color (src/store/reducers.js:14:26)
      at Object.<anonymous> (__tests__/store/reducers/color.test.js:43:36)
      at process._tickCallback (internal/process/next_tick.js:103:7)

color Reducer

  ✓ ADD_COLOR success (3ms)
  ✓ RATE_COLOR success (3ms)

Test Suites: 1 failed, 1 total
Tests:       1 failed, 1 passed, 2 total
Snapshots:  0 total
Time:        0.513s, estimated 1s
```

```
Ran all test suites.
```

Внесем изменения в преобразователь цвета с целью успешного выполнения теста. Воспользуемся оператором распространения, чтобы получить копию объекта состояния перед перезаписью рейтинга:

```
case 'RATE_COLOR':
  return {
    ...state,
    rating: action.rating
  }
```

Теперь, поскольку состояние изменению не подверглось, должны быть пройдены оба теста:

```
$ jest
```

```
PASS __tests__/store/reducers/color.test.js

color Reducer

  ✓ ADD_COLOR success (3ms)
  ✓ RATE_COLOR success

Test Suites: 1 passed, 1 total
Tests:       2 passed, 2 total
Snapshots:  0 total
Time:        0.782s, estimated 1s
```

```
Ran all test suites.
```

Описанный процесс представляет собой обычный цикл TDD-разработки. Сначала были созданы тесты, затем код, чтобы эти тесты были пройдены, после чего была выполнена реструктуризация как кода, так и тестов. Такой подход весьма эффективен при работе с JavaScript и особенно с Redux.

Тестирование хранилища

Работоспособность хранилища создает хорошие предпосылки для работоспособности всего приложения. Процесс тестирования хранилища включает создание хранилища с преобразователями, ввод предполагаемого состояния, диспетчеризацию действий и проверку результатов.

При тестировании хранилища можно объединить создатели действий и убить сразу двух зайцев, протестировав вместе хранилище и создатели.

В главе 8 была создана фабрика хранилища `storeFactory`, представлявшая собой функцию, которой можно воспользоваться для управления процессом создания хранилища в приложении органайзера цветов:

```
import { createStore,
        combineReducers,
        applyMiddleware } from 'redux'
import { colors, sort } from './reducers'
import stateData from '../data/initialState'

const logger = store => next => action => {
  let result
  console.groupCollapsed("dispatching", action.type)
  console.log('prev state', store.getState())
  console.log('action', action)
  result = next(action)
  console.log('next state', store.getState())
  console.groupEnd()
  return result
}

const saver = store => next => action => {
  let result = next(action)
  localStorage['redux-store'] = JSON.stringify(store.getState())
  return result
}

const storeFactory = (initialState=stateData) =>
  applyMiddleware(logger, saver)(createStore)(
    combineReducers({colors, sort}),
    (localStorage['redux-store']) ?
      JSON.parse(localStorage['redux-store']) :
      initialState
  )

export default storeFactory
```

Данный модуль экспортирует функцию, которую можно применить для создания хранилищ. При этом мы абстрагируемся от подробностей создания хранилища для органайзера цветов. В приведенном файле содержатся преобразователи, промежуточный код и исходное состояние, то есть все необходимое для создания хранилища для нашего приложения. При создании хранилища с помощью `storeFactory` можно дополнительно передать исходное состояние для нового хранилища, что поможет нам, когда придет время его протестировать.

В среде Jest имеются средства, позволяющие выполнять любой код до и после проведения каждого теста или каждого набора тестов. Функции `beforeAll` и `afterAll` вызываются соответственно до и после выполнения каждого набора тестов, а `beforeEach` и `afterEach` — до и после выполнения каждой инструкции `it`.



Функции, вызываемые до и после выполнения тестов

Укоренившейся практикой при написании тестов является использование в каждом тесте только одного утверждения¹. То есть в одной инструкции `it` не нужно многократно вызывать expect. Тогда каждое утверждение можно проверить по отдельности, что облегчает обнаружение причин неудач в прохождении тестов.

Для практической реализации такой схемы стоит применить функции, вызываемые в Jest до и после выполнения тестов. Выполняйте свой код тестов в `beforeAll` и проверяйте результаты с помощью нескольких инструкций `it`.

Посмотрим, как в файле `./__tests__/actions-spec.js` можно проверить хранилище в ходе тестирования создателя действия `addColor`. В следующем примере наше хранилище будет протестировано путем диспетчеризации создателя действия `addColor` и проверки результатов:

```
import C from '../src/constants'
import storeFactory from '../src/store'
import { addColor } from '../src/actions'

describe("addColor", () => {
  let store
  const colors = [
    {
      id: "8658c1d0-9eda-4a90-95e1-8001e8eb6036",
      title: "lawn",
      color: "#44ef37",
      timestamp: "Mon Apr 11 2016 12:54:19 GMT-0700 (PDT)",
      rating: 4
    },
  ],
  {
```

¹ См. публикацию Джея Фильдса (Jay Fields) `Testing: One Assertion per Test` по адресу <http://blog.jayfields.com/2007/06/testing-one-assertion-per-test.html> от 6 июня 2007 года.

```

        id: "f9005b4e-975e-433d-a646-79df172e1dbb",
        title: "ocean blue",
        color: "#0061ff",
        timestamp: "Mon Apr 11 2016 12:54:31 GMT-0700 (PDT)",
        rating: 2
      },
      {
        id: "58d9caee-6ea6-4d7b-9984-65b145031979",
        title: "tomato",
        color: "#ff4b47",
        timestamp: "Mon Apr 11 2016 12:54:43 GMT-0700 (PDT)",
        rating: 0
      }
    ]

beforeAll(() => {
  store = storeFactory({colors})
  store.dispatch(addColor("Dark Blue", "#000033"))
})

it("should add a new color", () =>
  expect(store.getState().colors.length).toBe(4))

it("should add a unique guid id", () =>
  expect(store.getState().colors[3].id.length).toBe(36))

it("should set the rating to 0", () =>
  expect(store.getState().colors[3].rating).toBe(0))

it("should set timestamp", () =>
  expect(store.getState().colors[3].timestamp).toBeDefined())
})

```

В целях создания экземпляра нового хранилища, содержащего в состоянии три образца, мы настраиваем тест на использование `storeFactory`. Затем проводим диспетчеризацию нашего создателя действия `addColor`, чтобы добавить к состоянию четвертый цвет под названием `dark blue`.

Теперь в каждом тесте проверяются результаты диспетчеризации действия. И каждый содержит одну инструкцию `expect`. Если некий тест не будет пройден, то мы точно будем знать, в каком из полей нового действия возникла проблема.

На этот раз используются два новых обнаружителя совпадений: `.toBe` и `toBeDefined`. Первый сравнивает результаты с помощью оператора `===`. Данный обнаружитель может применяться для сравнения примитивов, таких как числа или строки, а `.toEqual` служит для углубленного сравнения объектов. Обнаружитель совпадений `.toBeDefined` может использоваться для проверки наличия переменной или функции. В данном тесте ведется проверка наличия `timestamp`.

Эти тесты проверяют способность нашего хранилища успешно добавлять новые цвета с помощью создателей действий, что поможет убедиться в работоспособности нашего кода.

Тестирование компонентов React

Компоненты React предоставляют инструкции, которых библиотека должна придерживаться при создании и управлении обновлениями DOM. Эти компоненты могут тестироваться путем их отображения и проверки получающейся в результате DOM.

Наши тесты запускаются не в браузере, а в терминале с помощью Node.js. В этой среде нет API DOM, ставшего стандартом для каждого браузера. В Jest включен npm-пакет `jsdom`, который используется для имитации в Node.js браузерной среды, играющей в тестировании компонентов React весьма важную роль.

Настройка среды Jest

Jest дает возможность перед активизацией любых тестов запускать сценарий, где можно настроить дополнительные глобальные переменные, которые могут использоваться в любом из наших тестов.

Допустим, нужно добавить React к глобальной области видимости вместе с несколькими пробными цветами, доступными любому из наших тестов. Можно создать файл `__tests__/global.js`:

```
import React from 'react'
import deepFreeze from 'deep-freeze'

global.React = React
global._testColors = deepFreeze([
  {
    id: "8658c1d0-9eda-4a90-95e1-8001e8eb6036",
    title: "lawn",
    color: "#44ef37",
    timestamp: "Sun Apr 10 2016 12:54:19 GMT-0700 (PDT)",
    rating: 4
  },
  {
    id: "f9005b4e-975e-433d-a646-79df172e1dbb",
    title: "ocean blue",
    color: "#0061ff",
    timestamp: "Mon Apr 11 2016 12:54:31 GMT-0700 (PDT)",
    rating: 2
  },
  {
    id: "58d9caee-6ea6-4d7b-9984-65b145031979",
    title: "tomato",
    color: "#ff4b47",
    timestamp: "Fri Apr 15 2016 12:54:43 GMT-0700 (PDT)",
    rating: 0
  }
])
```

Данный файл добавляет в глобальную область видимости React и несколько неизменяющихся тестов цветов. Затем среде Jest следует сообщить о необходимости запуска данного файла перед стартом наших тестов. Это можно сделать, добавив к узлу `jest` в `package.json` поле `setupFiles`:

```
"jest": {
  "setupFiles": ["/__tests__/global.js"],
  "modulePathIgnorePatterns": ["global.js"]
}
```

Поле `setupFiles` используется для предоставления массива файлов, которые среда Jest должна запускать для установки глобальной среды перед активизацией наших тестов. Поле `modulePathIgnorePatterns` заставляет Jest проигнорировать при запуске тестов файл `global.js`, так как он является настроечным и не содержит набор тестов. Необходимость этого поля обусловлена следующим соображением: предпочтительнее добавлять файл `global.js` к папке `__tests__`, несмотря даже на то, что никакие тесты в нем не содержатся.

Игнорирование импорта SCSS. Если вы импортируете файлы SCSS (или CSS, или SASS) непосредственно в ваши компоненты, то на период тестирования их импортирование лучше проигнорировать. В противном случае они мешают успешному прохождению тестов.

Эти файлы можно проигнорировать путем введения отображателя модулей, возвращающего при импорте файлов `.css`, `.scss` или `.less` пустую строку. Установим пакет `jest-css-modules`:

```
npm install jest-css-modules --save-dev
```

Затем нужно указать среде Jest на необходимость использования данного модуля вместо любого `scss`-импорта. Для этого к узлу `jest` файла `package.json` следует добавить поле `moduleNameMapper`:

```
"jest": {
  "setupFiles": ["/__tests__/global.js"],
  "modulePathIgnorePatterns": ["global.js"],
  "moduleNameMapper": {
    "\\.(scss)$": "<rootDir>/node_modules/jest-css-modules"
  }
}
```

Данное поле заставит среду Jest вместо любого импорта с окончанием `.scss` использовать модуль `jest-css-modules`. Добавление этих строк кода к вашему файлу `package.json` предотвратит сбой тестов по причине `scss`-импорта.

Enzyme

Мы уже практически готовы приступить к тестированию наших компонентов React. Перед созданием нашего первого теста компонентов осталось установить еще два `npm`-модуля:

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


Enzyme (<http://airbnb.io/enzyme/>) является утилитой тестирования компонентов React, разработанной в команде Airbnb. Для работы Enzyme требуется `react-addons-test-utils`, инструментарий, который может использоваться для отображения компонентов и взаимодействия с ними в ходе тестирования. Кроме того, необходим `react-dom`, но мы предполагаем, что этот модуль уже установлен.

Enzyme облегчает отображение компонента на экране и обход выводимых на экран элементов. Enzyme не является средой тестирования или утверждения. Эта утилита выполняет задачу отображения компонентов React на экране для тестирования и предоставляет инструменты, необходимые для обхода дочерних элементов, проверки свойств и состояния, имитации событий и отправки запросов к DOM.

В Enzyme имеются три основных метода отображения.

- ❑ Метод `shallow` отображает в целях блочного тестирования компоненты не глубже одного уровня.
- ❑ Метод `mount` отображает компоненты с помощью DOM браузера и применяется для тестирования всего жизненного цикла компонента, а также свойств или состояния дочерних элементов.
- ❑ Метод `render` используется для отображения с компонентом статической разметки HTML. Задействуя `render`, можно проверить, что ваш компонент возвращает соответствующий код HTML.

Рассмотрим компонент `Star`:

```
const Star = ({ selected=false, onClick=f=>f }) =>
  <div className={(selected) ? "star selected" : "star"}
    onClick={onClick}>
  </div>
```

Он должен отобразить элемент `div` с помощью `className`, зависящего от выбранного свойства. Он должен также откликаться на события щелчков кнопкой мыши.

Напишем тест для компонента `Star` с применением Enzyme. Мы воспользуемся этой утилитой, чтобы отобразить компонент и найти конкретные элементы DOM в отображенном компоненте `Star`. Чтобы отобразить наш компонент на один уровень глубже, можно задействовать метод `shallow`:

```
import { shallow } from 'enzyme'
import Star from '../src/components/ui/Star'
```

```
describe("<Star /> UI Component", () => {
  it("renders default star", () =>
    expect(
      shallow(<Star />)
        .find('div.star')
        .length
    ).toBe(1)
  )
})
```

```
it("renders selected stars", () =>
  expect(
    shallow(<Star selected={true} />)
      .find('div.selected.star')
      .length
  ).toBe(1)
)
```

Enzyme поставляется с функциями, которые слегка напоминают функции jQuery. Метод `find` можно применять для запроса получающейся DOM, используя синтаксис селектора.

В первом тесте отображается примерный компонент `Star` и проверяется, получается ли в результате DOM, которая содержит элемент `div`, имеющий класс `star`. Во втором тесте отображается примерный выбранный компонент `Star` и проверяется, получается ли в результате DOM, которая содержит элемент `div`, имеющий как класс `star`, так и класс `selected`. Проверка длины позволяет убедиться, что в каждом тесте отображается только один `div`.

Затем следует протестировать событие щелчка. В Enzyme имеются средства, позволяющие имитировать события и проверять их выдачу. Для данного теста нам нужна функция, которую можно использовать для проверки работоспособности всего, что связано со свойством `onClick`. Нам нужна функция-имитатор, уже предусмотренная для нас в среде Jest:

```
it("invokes onClick", () => {
  const _click = jest.fn()

  shallow(<Star onClick={_click} />)
    .find('div.star')
    .simulate('click')

  expect(_click).toBeCalled()
})
```

В этом тесте с помощью `jest.fn` создается функция-имитатор под названием `_click`. При отображении `Star` мы отправляем данную функцию в качестве свойства `onClick`. Затем находим отображенный элемент `div` и имитируем щелчок кнопкой мыши на данном элементе, задействуя имеющийся в Enzyme метод `simulate`. Щелчок на `Star` должен вызвать свойство `onClick`, а это, в свою очередь, должно привести к вызову нашей функции-имитатора. Для проверки факта вызова функции можно использовать обнаружитель совпадений `.toBeCalled`.

Утилита Enzyme может применяться для отображения компонентов, нахождения отображенных элементов DOM или других компонентов и в работе по взаимодействию с ними.

Имитация компонентов

В последнем тесте мы представили концепцию имитации: для тестирования компонента `Star` воспользовались функцией-имитатором. Среда Jest располагает полноценным набором инструментов, способствующих созданию и вставке разнообразных имитаторов, помогающих писать высококачественные тесты. Имитация — весьма важная технология тестирования, позволяющая сосредоточиться на блочном тестировании. Имитаторы представляют собой объекты, применяемые вместо реальных объектов в целях тестирования¹.

Имитаторы для мира тестирования являются тем же самым, что дублеры-каскадеры для Голливуда. И имитаторы, и дублеры используются вместо реальных персонажей (компонентов или кинозвезд). В фильме дублер похож на настоящего актера. А в тесте объект-имитатор похож на реальный объект.

Цель имитации — позволить вам сосредоточиться на тестах какого-либо проверяемого компонента или объекта, то есть на тестировании SUT. Имитаторы используются вместо объектов, компонентов или функций, от которых зависит ваша SUT, и дают возможность убедиться, что она работает должным образом без каких-либо помех со стороны своих зависимостей. Имитация позволяет изолировать, создать и протестировать функциональность независимо от других компонентов.

Тестирование компонентов высшего порядка

Одной из областей, где будут применяться имитаторы, станет тестирование компонентов высшего порядка (НОС). Данные компоненты отвечают за придание функциональных возможностей вставленным компонентам через свойства. Мы можем создать компонент-имитатор и отправить его НОС, чтобы убедиться, что тот добавляет соответствующие свойства нашему имитатору.

Взглянем на тест для `Expandable`, НОС, разработанного ранее в главе 7. Для настройки теста НОС сначала нужно создать компонент-имитатор и отправить его НОС. Дублером, используемым вместо реального компонента, станет `MockComponent`:

```
import { mount } from 'enzyme'
import Expandable from '../src/components/НОС/Expandable'

describe("Expandable Higher-Order Component", () => {
  let props,
      wrapper,
      ComposedComponent,
```

¹ Углубленное представление об имитаторах можно получить, прочитав статью Мартина Фаулера (Martin Fowler) `Mocks Aren't Stubs` по адресу <https://martinfowler.com/articles/mocksArentStubs.html>.

```

    MockComponent = ({collapsed, expandCollapse}) =>
      <div onClick={expandCollapse}>
        {(collapsed) ? 'collapsed' : 'expanded'}
      </div>

    describe("Rendering UI", ... )

    describe("Expand Collapse Functionality", ... )

  })

```

`MockComponent` является простым функциональным, не имеющим состояния, разработанным динамически компонентом. Он возвращает элемент `div` с обработчиком события щелчка кнопкой мыши `onClick`, который будет использоваться для тестирования функции `expandCollapse`. В компоненте-имитаторе также показывается нахождение в состоянии показа (`expanded`) или в состоянии скрытия (`collapsed`). Кроме как в этом тесте, данный компонент больше нигде применяться не будет.

Тестируемой системой является НОС `Expandable`. Перед тестом мы вызовем этот компонент с помощью нашего имитатора и проверим возвращенный компонент на предмет применения к нему соответствующих свойств.

Вместо функции `shallow` мы воспользуемся функцией `mount`, следовательно, получим возможность проверить свойства и состояние возвращенного компонента:

```

describe("Rendering UI", () => {
  beforeAll(() => {
    ComposedComponent = Expandable(MockComponent)
    wrapper = mount(<ComposedComponent foo="foo" gnar="gnar"/>)
    props = wrapper.find(MockComponent).props()
  })

  it("starts off collapsed", () =>
    expect(props.collapsed).toBe(true)
  )

  it("passes the expandCollapse function to composed component", () =>
    expect(typeof props.expandCollapse)
      .toBe("function")
  )

  it("passes additional foo prop to composed component", () =>
    expect(props.foo)
      .toBe("foo")
  )

  it("passes additional gnar prop to composed component", () =>
    expect(props.gnar)
      .toBe("gnar")
  )
})

```

Как только с помощью НОС будет создан составной компонент, можно проверить, что составной компонент добавил соответствующие свойства к нашему компоненту-имитатору путем его подключения и непосредственной проверки объекта свойств. Этот тест позволяет убедиться, что НОС добавил свойство `collapsed` и метод для его изменения под названием `expandCollapse`. Он также дает возможность отследить, что любые свойства, добавленные к составному компоненту, `foo` и `gnar`, занимают свое место в имитаторе.

А теперь проверим возможность изменения свойства `collapsed` нашего составного компонента:

```
describe("Expand Collapse Functionality", () => {
  let instance

  beforeAll(() => {
    ComposedComponent = Expandable(MockComponent)
    wrapper = mount(<ComposedComponent collapsed={false}/>)
    instance = wrapper.instance()
  })

  it("renders the MockComponent as the root element", () => {
    expect(wrapper.first().is(MockComponent))
  })

  it("starts off expanded", () => {
    expect(instance.state.collapsed).toBe(false)
  })

  it("toggles the collapsed state", () => {
    instance.expandCollapse()
    expect(instance.state.collapsed).toBe(true)
  })
})
```

После подключения компонента появляется возможность сбора информации об отображенном экземпляре с помощью метода `wrapper.instance`. В данном случае нам нужно, чтобы компонент запускался скрытым. Для подтверждения факта запуска экземпляра компонента в скрытом состоянии можно проверить как его свойства, так и состояние.

У объекта-контейнера `wrapper` также имеются методы для обхода элементов DOM-модели. В первом тестовом сценарии выбирается первый дочерний элемент, для чего используется метод `wrapper.first`; после этого проверяется, является ли данный элемент экземпляром нашего компонента `MockComponent`.

НОС отлично подходят для представления имитаторов, поскольку процесс вставки имитатора не составляет труда: нужно просто отправить его НОС в качестве аргумента. Аналогично выглядит и концепция имитации закрытых компонентов, но внедрение происходит немного сложнее.

Jest-имитаторы

Среда Jest позволяет вставлять имитаторы не только в НОС, но и в любой наш компонент. Применяя Jest, можно симитировать любой модуль, импортируемый вашей SUT. Имитация позволяет сконцентрировать тестирование на SUT, а не на других модулях, которые могут быть потенциальными источниками проблем.

Посмотрим, к примеру, на компонент `ColorList`, импортирующий компонент `Color`:

```
import { PropTypes } from 'react'
import Color from './Color'
import './../../../../stylesheets/ColorList.scss'

const ColorList = ({ colors=[], onRate=f=>f, onRemove=f=>f }) =>
  <div className="color-list">
    {(colors.length === 0) ?
      <p>No Colors Listed. (Add a Color)</p> :
      colors.map(color =>
        <Color key={color.id}
            {...color}
            onRate={(rating) => onRate(color.id, rating)}
            onRemove={() => onRemove(color.id)} />
      )
    }
  </div>

ColorList.propTypes = {
  colors: PropTypes.array,
  onRate: PropTypes.func,
  onRemove: PropTypes.func
}

export default ColorList
```

Мы должны убедиться в правильном функционировании компонента `ColorList`. И нас совершенно не интересует компонент `Color`, которому следует иметь собственный блочный тест. Для `ColorList` можно написать тест, заменяющий компонент `Color` имитатором:

```
import { mount } from 'enzyme'
import ColorList from './../../../../src/components/ui/ColorList'

jest.mock('./../../../../src/components/ui/Color', () =>
  ({rating, onRate=f=>f}) =>
    <div className="mock-color">
      <button className="rate" onClick={() => onRate(rating)} />
    </div>
)

describe("<ColorList /> UI Component", () => {
  describe("Rating a Color", () => {
```

```

let _rate = jest.fn()

beforeAll(() =>
  mount(<ColorList colors={_testColors} onRate={_rate} />)
    .find('button.rate')
    .first()
    .simulate('click')
)

it("invokes onRate Handler", () =>
  expect(_rate).toBeCalled()
)

it("rates the correct color", () =>
  expect(_rate).toBeCalledWith(
    "8658c1d0-9eda-4a90-95e1-8001e8eb6036",
    4
  )
)
})
})

```

В данном тесте для вставки имитатора вместо настоящего компонента `Color` используется метод `jest.mock`. Первый аргумент, отправляемый в адрес `jest.mock`, — модуль, который нам нужно симитировать, а второй — функция, возвращающая симитированный компонент. В данном случае имитатор `Color` представляет собой урезанную версию компонента `Color`. Единственной заботой теста является оценка цвета, поэтому имитатор работает только со свойствами оценки такого образца.

В момент запуска данного теста среда Jest заменит компонент `Color` нашим имитатором. При отображении `ColorList` мы отправляем ранее глобально определенные в данной главе пробные цвета `_testColors`. Когда `ColorList` отображает каждый цвет, вместо него на экране будет показан наш имитатор. Когда имитируется событие щелчка на первой кнопке, оно происходит с нашим первым имитатором.

Отображенная DOM для этого компонента будет выглядеть так:

```

<ColorList>
  <div className="color-list">
    <MockColor onRate=[[Function]] rating={4}>
      <div className="mock-color">
        <button id="rate" onClick=[[Function]] />
      </div>
    </MockColor>
    <MockColor onRate=[[Function]] rating={2}>
      <div className="mock-color">
        <button id="rate" onClick=[[Function]] />
      </div>
    </MockColor>
    <MockColor onRate=[[Function]] rating={0}>
      <div className="mock-color">

```

```

        <button id="rate" onClick={{[Function]}} />
      </div>
    </MockColor>
  </div>
</ColorList>

```

Настоящий компонент `Color` передаст выбранную оценку компоненту `ColorList`, а наш имитатор не использует оценочный компонент `StarRating`. Он не занимается оценкой цветов; вместо этого он притворяется оценщиком цвета, просто возвращая текущую оценку компоненту `ColorList`. В данном тесте компонент `Color` нас не интересует, в сферу нашего внимания входит только `ColorList`. Этот компонент ведет себя ожидаемым образом. В результате щелчка на первом цвете свойству `onRate` передается правильная оценка.

Указываемые имитаторы

Для использования в имитаторах среда Jest позволяет создавать модули. Вместо добавления кода для имитаторов непосредственно тестам нужно поместить каждый имитатор в его собственный файл в папке `__mocks__`, где Jest станет их искать.

Посмотрим на файл `/src/components/containers.js`, который мы создали в главе 9. Он состоит из трех контейнеров. Для нашего следующего теста сконцентрируемся на контейнере `Colors`:

```

import ColorList from './ui/ColorList'

export const Colors = connect(
  state =>
    ({
      colors: [...state.colors].sort(sortFunction(state.sort))
    }),
  dispatch =>
    ({
      onRemove(id) {
        dispatch(removeColor(id))
      },
      onRate(id, rating) {
        dispatch(rateColor(id, rating))
      }
    })
)(ColorList)

```

Контейнер `Colors` используется для подключения данных из хранилища к компоненту `ColorList`. Он сортирует цвета, найденные в состоянии, и отправляет их компоненту `ColorList` в виде свойства. Он также обрабатывает свойства функций `onRate` и `onRemove`, имеющихся в `ColorList`. И наконец, этот контейнер зависит от модуля `ColorList`.

Указываемый имитатор создается путем добавления файла `<Module>.js` к папке `__mocks__`. В данной папке содержатся имитируемые модули, используемые в ходе тестирования вместо настоящих модулей.

Например, к текущему проекту мы добавим имитатор `ColorList`, для чего в папке `/src/components/ui`, на том же самом уровне, что и компонент `ColorList`, создадим папку `__mocks__`. Затем в эту папку поместим наш имитатор `ColorList.js`.

Данный имитатор будет просто отображать пустой элемент `div`. Посмотрите на код для имитатора `ColorList.js`:

```
const ColorListMock = () => <div className="color-list-mock"></div>

ColorListMock.displayName = "ColorListMock"

export default ColorListMock
```

Теперь при имитации компонента `/src/components/ui/ColorList` с помощью метода `jest.mock` среда Jest станет брать соответствующий имитатор из папки `__mocks__`. И нам не нужно будет определять имитатор непосредственно в тесте.

Кроме указываемой имитации `ColorList`, мы создаем имитатор и для хранилища. Напомним, что хранилища выполняют три важные функции: диспетчеризации — `dispatch`, подписки — `subscribe` и получения состояния — `getState`. Наша имитация хранилища тоже будет их выполнять. Функция `getState` предоставляет реализацию для этой функции-имитации, возвращающей образец состояния с помощью глобальных тестовых цветов.

Данным имитатором хранилища мы воспользуемся для тестирования контейнера. Компонент `Provider` будет отображаться с имитатором хранилища в виде свойства `store`. Наш контейнер должен получать цвета из хранилища, сортировать их и отправлять имитатору:

```
import { mount, shallow } from 'enzyme'
import { Provider } from 'react-redux'
import { Colors } from '.././././src/components/containers'

jest.mock('.././././src/components/ui/ColorList')

describe("<Colors /

  let wrapper

  const _store = {
    dispatch: jest.fn(),
    subscribe: jest.fn(),
    getState: jest.fn(() =>
      ({
        sort: "SORTED_BY_DATE",
        colors: _testColors
      })
    )
  }

  beforeAll(() => wrapper = mount(
    <Provider store={_store}>
```

```

        <Colors />
    </Provider>
  ))

  it("renders three colors", () => {
    expect(wrapper
      .find('ColorListMock')
      .props()
      .colors
      .length
    ).toBe(3)
  })

  it("sorts the colors by date", () => {
    expect(wrapper
      .find('ColorListMock')
      .props()
      .colors[0]
      .title
    ).toBe("tomato")
  })
})

```

В этом тесте `jest.mock` вызывается для имитации компонента `ColorList`, но ему отправляется только один аргумент: путь к имитируемому модулю. Среда Jest знает, что реализацию данного имитатора нужно искать в папке `__mocks__`. Настоящий `ColorList` больше не используется, мы применяем урезанный компонент-имитатор. После отображения наша DOM должна выглядеть следующим образом:

```

<Provider>
  <Connect(ColorListMock)>
    <ColorListMock colors={[...]}
      onRate={{[Function]}}
      onRemove={{[Function]}}>
      <div className="color-list-mock" />
    </ColorListMock>
  </Connect(ColorListMock)>
</Provider>

```

Если наш контейнер работает, то отправит имитатору три цвета. Контейнер должен отсортировать их по дате. Проверить это можно, удостоверившись, что первым цветом является `tomato`, поскольку из трех цветов в `_testColors` у него самая свежая временная метка.

Добавим еще несколько тестов, чтобы убедиться в надлежащей работе `onRate` и `onRemove`:

```

afterEach(() => jest.resetAllMocks())

it("dispatches a REMOVE_COLOR action", () => {
  wrapper.find('ColorListMock')
    .props()

```

```
    .onRemove('f9005b4e-975e-433d-a646-79df172e1dbb')
  expect(_store.dispatch.mock.calls[0][0])
    .toEqual({
      id: 'f9005b4e-975e-433d-a646-79df172e1dbb',
      type: 'REMOVE_COLOR'
    })
})

it("dispatches a RATE_COLOR action", () => {
  wrapper.find('ColorListMock')
    .props()
    .onRate('58d9caee-6ea6-4d7b-9984-65b145031979', 5)
  expect(_store.dispatch.mock.calls[0][0])
    .toEqual({
      id: "58d9caee-6ea6-4d7b-9984-65b145031979",
      type: "RATE_COLOR",
      rating: 5
    })
})
```

Для тестирования `onRate` и `onRemove` нам не нужно имитировать щелчки кнопкой мыши. Достаточно вызвать соответствующие функциональные свойства с определенной информацией и проверить, что имеющийся у хранилища метод `dispatch` был вызван с правильными данными. Вызов свойства `onRemove` должен заставить хранилище провести диспетчеризацию действия `REMOVE_COLOR`, а вызов свойства `onRate` — диспетчеризацию действия `RATE_COLOR`. Кроме того, нужно убедиться в перезапуске имитатора `dispatch` после завершения каждого теста.

Возможность простого внедрения имитаторов в тестируемые модули является одной из самых впечатляющих особенностей среды Jest. Имитация — весьма эффективная технология, позволяющая сконцентрировать тесты на SUT.

Тестирование на основе отображения мгновенного состояния (Snapshot Testing)

Разработка, основанная на тестировании (TDD), предоставляет широкие возможности для тестирования вспомогательных функций, пользовательских классов и наборов данных. Но как только дело доходит до тестирования UI, TDD усложняется и зачастую становится непрактичной. Пользовательский интерфейс подвержен частым изменениям, что превращает сопровождение его тестов в весьма трудоемкую процедуру. Кроме того, часто ставится задача создания тестов для компонентов UI, уже применяемых на практике.

Snapshot-тестирование предоставляет способ быстро проверить компоненты пользовательского интерфейса и убедиться в отсутствии каких-либо неожиданных изменений. Среда Jest может сохранять мгновенное состояние отображенного UI

и сравнивать его с результатами последующих тестов. Тем самым мы получаем возможность проверить отсутствие каких-либо неожиданных эффектов после обновлений, сохраняя при этом высокие темпы работы, не застревая на практическом тестировании пользовательского интерфейса. Кроме того, при ожидаемых изменениях UI отображения мгновенного состояния можно легко обновить.

Посмотрим, как можно протестировать компонент `Color` с помощью технологии snapshot-тестирования. Сначала взглянем на уже имеющийся код этого компонента:

```
import { PropTypes, Component } from 'react'
import StarRating from './StarRating'
import TimeAgo from './TimeAgo'
import FaTrash from 'react-icons/lib/fa/trash-o'
import '.././././stylesheets/Color.scss'

class Color extends Component {

  render() {
    const {
      title, color, rating, timestamp, onRemove, onRate
    } = this.props

    return (
      <section className="color" style={this.style}>
        <h1 ref="title">{title}</h1>
        <button onClick={onRemove}>
          <FaTrash />
        </button>
        <div className="color"
          style={{ backgroundColor: color }}>
        </div>
        <TimeAgo timestamp={timestamp} />
        <div>
          <StarRating starsSelected={rating} onRate={onRate}/>
        </div>
      </section>
    )
  }
}

export default Color
```

При отображении этого компонента с конкретными свойствами мы вправе ожидать DOM, содержащую конкретные компоненты, основанные на отправленных свойствах:

```
shallow(
  <Color title="Test Color"
    color="#F0F0F0"
    rating={3}
    timestamp="Mon Apr 11 2016 12:54:19 GMT-0700 (PDT)"
  />
).html()
```

Получающаяся при этом DOM должна иметь следующий вид:

```
<section class="color">
  <h1>Test Color</h1>
  <button><svg /></button>
  <div class="color" style="background-color:#F0F0F0;"></div>
  <div class="time-ago">4/11/2016</div>
  <div>
    <div class="star-rating">
      <div class="star selected"></div>
      <div class="star selected"></div>
      <div class="star selected"></div>
      <div class="star"></div>
      <div class="star"></div>
      <p>3 of 5 stars</p>
    </div>
  </div>
</section>
```

Snapshot-тестирование позволит сохранить отображение мгновенного состояния этой DOM при самом первом запуске теста. Затем мы получим возможность сравнить с этим отображением результаты последующих тестов, чтобы убедиться в неизменности получающихся выводов на экран.

Приступим к созданию snapshot-теста для компонента `Color`:

```
import { shallow } from 'enzyme'
import Color from '../src/components/ui/Color'

describe("<Color /> UI Component", () => {
  it("Renders correct properties", () => {
    let output = shallow(
      <Color title="Test Color"
        color="#F0F0F0"
        rating={3}
        timestamp="Mon Apr 11 2016 12:54:19 GMT-0700 (PDT)"
      />
    ).html()

    expect(output).toMatchSnapshot()
  })
})
```

В этом тесте мы используем `Enzyme`, чтобы отобразить компонент и собрать получающийся вывод в строку кода HTML. Для snapshot-тестирования применяется имеющийся в среде `Jest` обнаружитель совпадений `.toMatchSnapshot`. При первом запуске данного теста среда `Jest` сохранит копию получившегося кода HTML в `snapshot`-файле. Он будет добавлен к папке `__snapshots__`, которая находится в том же каталоге, что и тест. На данный момент `snapshot`-файл будет иметь следующий вид:

```
exports[`<Color /> UI Component Renders correct properties 1`] =
  `<section class="color"><h1>Test Color</h1><button><svg ...`
```

При каждом последующем запуске теста среда Jest станет сравнивать вывод с одним и тем же snapshot-файлом. Малейшее отклонение в получающемся коде HTML повлечет неудачное завершение теста.

Snapshot-тестирование позволяет быстро продвигаться вперед, но если поспешить, то можно в конечном итоге создать *непредсказуемо выполняемые тесты*, или тесты, которые будут успешно пройдены в тех случаях, когда этого не должно происходить. Для тестирования получение snapshot-файлов в виде строк HTML вполне допустимо, но для нас самих проверка фактической приемлемости отображения мгновенного состояния окажется затруднена. Усовершенствуем snapshot-файл, сохранив вывод в формате JSX.

Для этого нужно установить модуль `enzyme-to-json`:

```
npm install enzyme-to-json --save-dev
```

Данный модуль предоставляет функцию, которой можно воспользоваться для вывода Enzyme-надстроек в виде JSX; это упростит проверку snapshot-вывода на приемлемость.

Чтобы вывести наш snapshot-образ с помощью `enzyme-to-json`, сначала выполним поверхностный вывод компонента `Color`, задействуя `Enzyme`, затем отправим получившийся результат функции `toJSON`, а уже результат данной функции отправим функции `expect`. При этом может появиться соблазн создать код следующего вида:

```
expect(
  toJSON(
    shallow(
      <Color title="Test Color"
        color="#F0F0F0"
        rating={3}
        timestamp="Mon Apr 11 2016 12:54:19 GMT-0700 (PDT)"
      />
    )
  )
).toMatchSnapshot()
```

Но в данном случае будет весьма уместно использовать небольшую композицию для улучшения кода. Помните, что такое композиция? Небольшие функции могут быть помещены вместе в целях составления более крупной функции. Для создания из `shallow`, `toJSON` и `expect` одной более крупной функции можно применить функцию `compose` из арсенала `Redux`:

```
import { shallow } from 'enzyme'
import toJSON from 'enzyme-to-json'
import { compose } from 'redux'
import Color from '.././././src/components/ui/Color'

describe("<Color /> UI Component", () => {

  const shallowExpect = compose(expect, toJSON, shallow)
```

```

it("Renders correct properties", () =>
  shallowExpect(
    <Color title="Test Color"
      color="#F0F0F0"
      rating={3}
      timestamp="Mon Apr 11 2016 12:54:19 GMT-0700 (PDT)"
    />
  ).toMatchSnapshot()
)
})

```

Функция `shallowExpect` получает компонент, производит его поверхностный вывод, преобразует результат в JSON-формат, а затем отправляет его методу `expect`, который возвращает все обнаружители совпадений среды Jest.

Если запустить этот тест, то он не будет пройден, поскольку вывод теперь имеет вид JSX-, а не HTML-строки. Наш тест уже не находит совпадения с `snapshot`-файлом. Но `snapshot`-файлы можно просто обновить, запустив тест еще раз с ключом `updateSnapshot`:

```
jest --updateSnapshot
```

Если запустить среду Jest с ключом `watch`:

```
jest --watch
```

то она продолжит работу в терминале, отслеживая при этом изменения в нашем исходном коде и тестах. При внесении каких-либо изменений среда Jest перезапустит тесты. В момент отслеживания тестов можно просто обновить `snapshot`-файл, нажав клавишу `U`:

```

Snapshot Summary
  > 1 snapshot test failed in 1 test suite. Inspect your code changes or press
  `u` to update them.

```

```

Test Suites: 1 failed, 6 passed, 7 total
Tests:      1 failed, 28 passed, 29 total
Snapshots:  1 failed, 1 total
Time:       1.407s
Ran all test suites.

```

Watch Usage

- > Press `u` to update failing snapshots.
- > Press `p` to filter by a filename regex pattern.
- > Press `q` to quit watch mode.
- > Press `Enter` to trigger a test run.

Как только `snapshot`-файл обновится, тест будет пройден. Теперь `snapshot`-файл изменился. Вместо длинной HTML-строки он выглядит следующим образом:

```

exports[`<Color /> UI Component Renders correct properties 1`] = `
<section
  className="color">

```

```

<h1>
  Test Color
</h1>
<button
  onClick={{[Function]}}
  <FaTrash0 />
</button>
<div
  className="color"
  style={
    Object {
      "backgroundColor": "#F0F0F0",
    }
  } />
<TimeAgo
  timestamp="Mon Apr 11 2016 12:54:19 GMT-0700 (PDT)" />
<div>
  <StarRating
    onRate={{[Function]}}
    starsSelected={3} />
</div>
</section>
';

```

Этот snapshot-файл читается гораздо легче. Его можно бегло просмотреть, чтобы проверить правильность результатов, прежде чем переходить к следующему тесту. Snapshot-тестирование может стать весьма эффективным способом быстрого добавления тестирования к вашим приложениям.

Использование данных об охвате кода

Выдача данных об охвате кода представляет собой процесс создания отчета о количестве фактически протестированных строк кода. Он предоставляет показатель, который может помочь в принятии решения о создании достаточного количества тестов.

Среда Jest поставляется с JavaScript-утилитой Istanbul, используемой для обзора тестов и создания отчета с описанием количества инструкций, ветвлений, функций и строк кода, охваченного тестированием.

Чтобы запустить Jest с выдачей данных об охвате кода, нужно просто при запуске команды `jest` добавить ключ `coverage`:

```
jest --coverage
```

В терминале будет создан и выведен на экран отчет о текущем охвате кода:

```

PASS __tests__/components/ui/ColorList.test.js
PASS __tests__/components/containers/Colors.test.js
PASS __tests__/components/ui/Color.test.js
PASS __tests__/components/ui/Star.test.js

```



```
PASS __tests__/components/HOC/Expandable.test.js
PASS __tests__/actions.test.js
PASS __tests__/store/reducers/color.test.js
```

File	% Stmts	% Branch	% Funcs	% Lines	Uncov'd Lines
All files	68.42	43.33	45.59	72.39	
src	100	100	100	100	
actions.js	100	100	100	100	
constants.js	100	100	100	100	
src/components	58.33	100	40	58.33	
containers.js	58.33	100	40	58.33	11,13,20,24,26
src/components/HOC	100	100	100	100	
Expandable.js	100	100	100	100	
src/components/ui	45.65	35.29	24	50	
AddColorForm.js	16.67	0	0	18.18	... 13,16,18,21
Color.js	66.67	100	33.33	66.67	40,41
ColorList.js	62.5	40	50	83.33	13
SortMenu.js	37.5	0	0	42.86	11,14,18,19
Star.js	100	100	100	100	
StarRating.js	33.33	0	0	40	5,7,9
TimeAgo.js	50	100	0	50	4
src/lib	58.54	15	16.67	67.65	
array-helpers.js	60	33.33	60	71.43	6,8
time-helpers.js	58.06	0	0	66.67	... 43,45,49,54
src/store	97.14	70	100	96.77	
index.js	100	100	100	100	
reducers.js	94.12	64.71	100	94.12	21

```
Test Suites: 7 passed, 7 total
Tests:       29 passed, 29 total
Snapshots:  1 passed, 1 total
Time:        1.691s, estimated 2s
```

Ran all test suites.

Watch Usage

- > Press p to filter by a filename regex pattern.
- > Press q to quit watch mode.
- > Press Enter to trigger a test run.

Данный отчет информирует о том, какой объем кода в каждом файле был выполнен в ходе тестирования, и представляет подотчет в отношении всех файлов, импортированных в тесты.

Среда Jest также создает отчет, который можно запустить в браузере, где представляются более подробные данные о коде, охваченном тестами. После запуска Jest с выдачей отчета об охвате можно заметить, что к корневому каталогу была добавлена папка `coverage`. Откройте в веб-браузере следующий файл: `/coverage/lcov-report/index.html`. Он покажет вам охват вашего кода в интерактивном отчете (рис. 10.1).

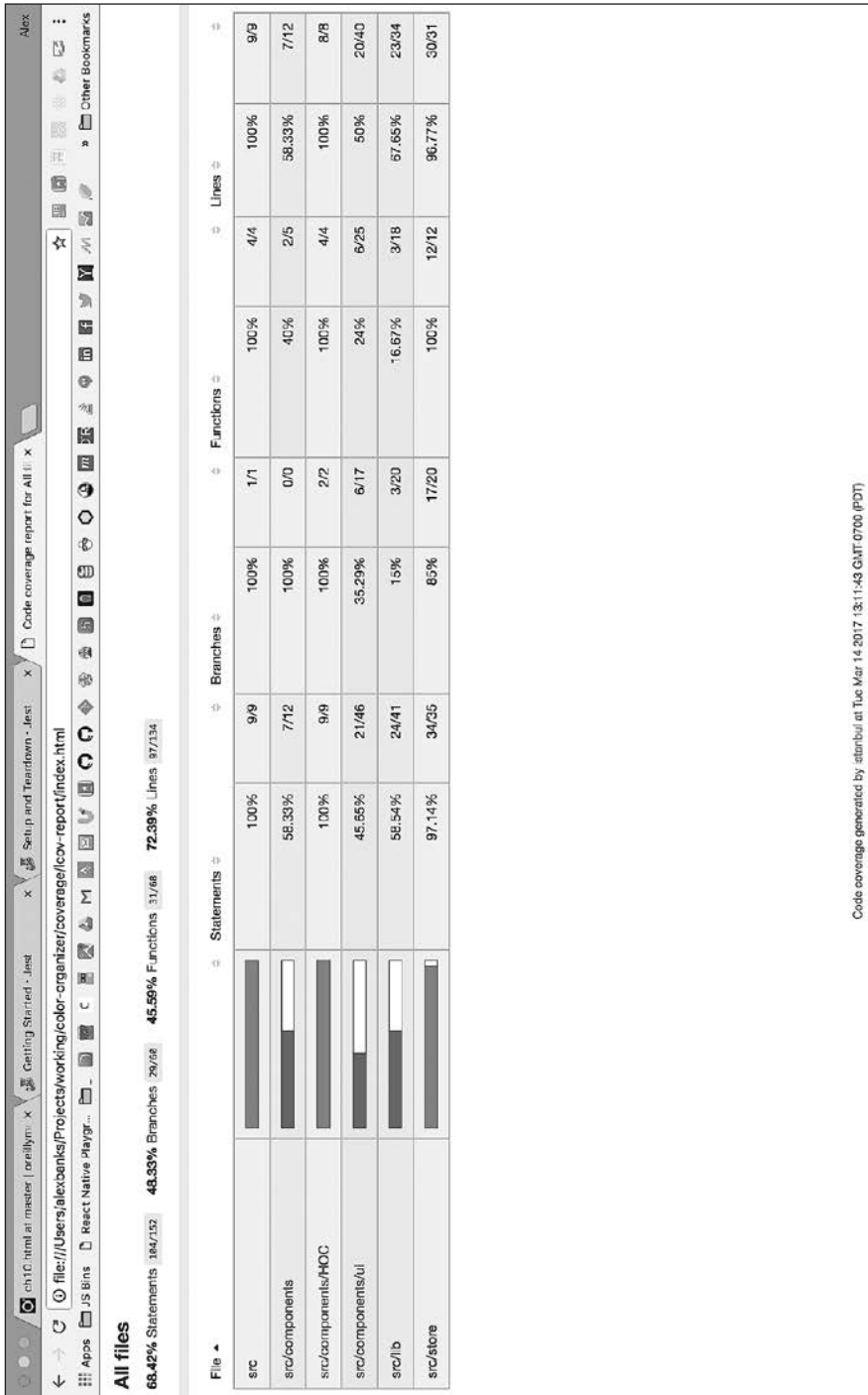


Рис. 10.1. Отчет об охвате кода тестированием

Из этого отчета можно узнать, какой объем кода был охвачен тестированием, а также об отдельных показателях охвата в каждой подпапке. Можно углубиться в подпапку и посмотреть, насколько плотно были охвачены тестированием отдельные файлы. Если выбрать папку `components/ui`, то станет видно, насколько плотно охвачены тестированием компоненты вашего пользовательского интерфейса (рис. 10.2).

Охват тестированием строк кода в отдельно взятом файле можно увидеть, щелкнув кнопкой мыши на имени файла. На рис. 10.3 показаны строки, охваченные тестированием в компоненте `ColorList`.

Компонент `ColorList` был протестирован вполне достаточно. В столбце в левой части экрана можно увидеть, сколько раз была выполнена в тесте каждая строка кода. Строки кода, выделенные желтым и красным цветом, не выполнялись. В данном случае похоже, что было протестировано лишь свойство `onRemove`. Добавим в набор тестов файла `ColorList.test.js` тест для свойства `onRemove` и охватим тестированием кодовую строку 13:

```
jest.mock('../src/components/ui/Color', () =>
  ({rating, onRate=f=>f, onRemove=f=>f}) =>
    <div className="mockColor">
      <button className="rate" onClick={() => onRate(rating)} />
      <button className="remove" onClick={onRemove} />
    </div>
)
..
describe("Removing a Color", () => {
  let _remove = jest.fn()

  beforeEach(() =>
    mount(<ColorList colors={_testColors} onRemove={_remove} />
      .find('button.remove')
      .last()
      .simulate('click')
    )

  it("invokes onRemove Handler", () =>
    expect(_remove).toBeCalled()
  )

  it("removes the correct color", () =>
    expect(_remove).toBeCalledWith(
      "58d9caee-6ea6-4d7b-9984-65b145031979"
    )
  )
})
```

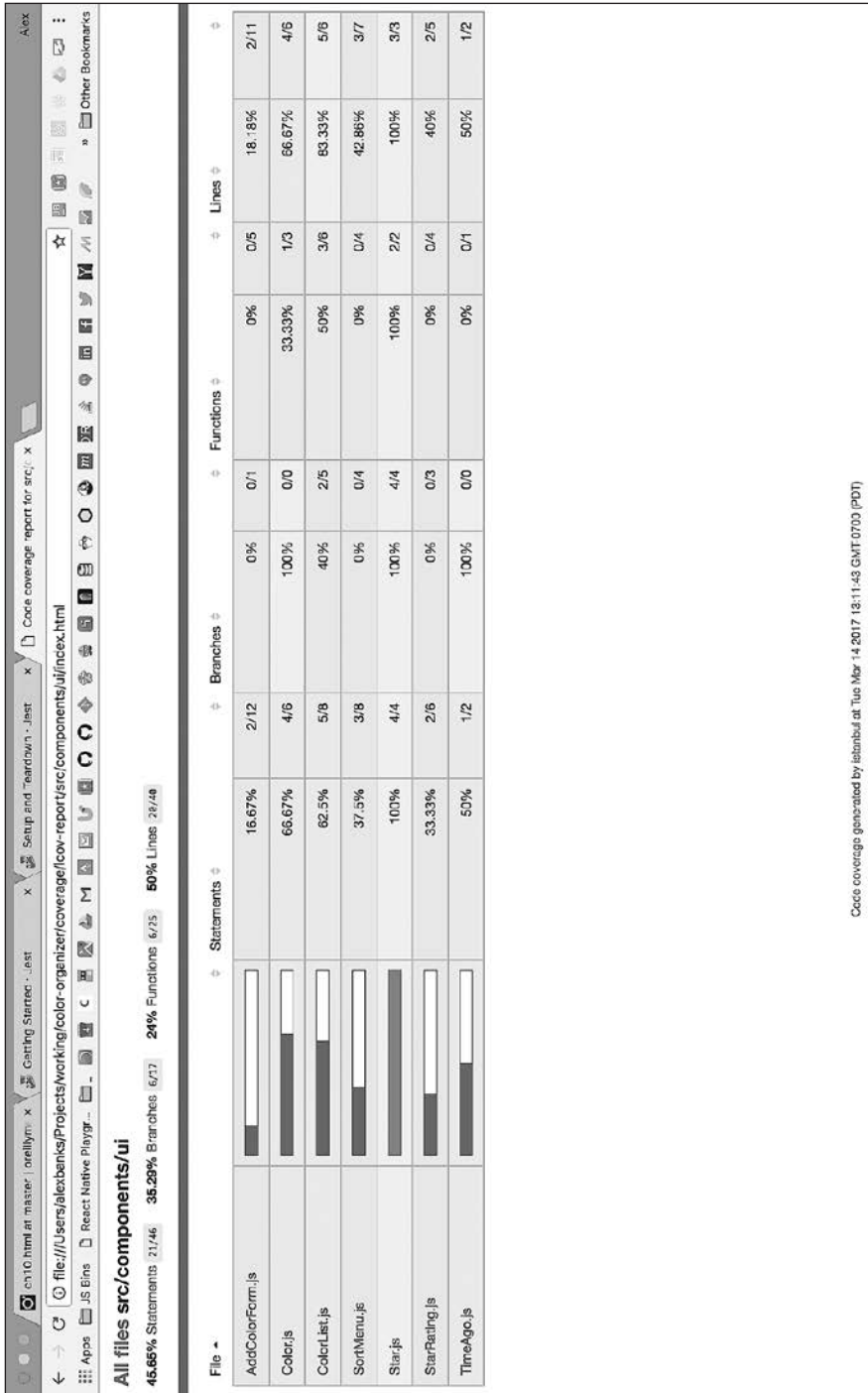


Рис. 10.2. Охват тестированием компонента пользовательского интерфейса

Code coverage report for src/ColorList.js.html

All files / src/components/ui/ColorList.js

62.5% Statements 5/8 40% Branches 7/5 50% Functions 3/6 83.33% Lines 5/6

```

1 import { PropTypes } from 'react'
2 import Color from './Color'
3 import './../styleSheets/ColorList.scss'
4
5 1x const ColorList = ({ colors=[], onRate=fn, onRemove=fn }) =>
6 1x {
7   1x <div className="color-list">
8     3x <span> Colors Listed. (Add a Color)</span> </>
9     3x <Color key={color.id}
10      3x color={color} =>
11        {...color}
12      1x onRate={rating} => onRate(color.id, rating)}
13      1x onRemove={() => onRemove(color.id)} />
14   1x </div>
15
16
17
18 1x ColorList.propTypes = {
19   1x colors: PropTypes.array,
20   1x onRate: PropTypes.func,
21   1x onRemove: PropTypes.func
22 }
23
24 export default ColorList

```

Code coverage generated by Istanbul at Tue Mar 14 2017 13:11:43 GMT-0700 (PDT)

Рис. 10.3. Охват тестированием компонента ColorList

К имитатору `Color` было добавлено свойство `onRemove`, а также кнопка для его запуска. При отображении `ColorList` данное свойство протестируется абсолютно тем же способом, которым проверялось свойство `onRate`. Мы выведем на экран компонент `ColorList` с нашими тремя цветами, нажмем последнюю кнопку удаления и убедимся в том, что функции-имитатору удаления `_remove` был передан правильный идентификатор цвета.

При следующем создании отчета об охвате кода тестированием станет заметно улучшение: теперь строка кода 13 показана охваченной тестированием (рис. 10.4).

Похоже, строка кода 8 тоже не была охвачена тестированием. Дело в том, что мы никогда не отображаем `ColorList` с пустым массивом цветов. Охватим тестированием и эту строку:

```
describe("Rendering UI", () => {
  it("Defaults properties correctly", () =>
    expect(shallow(<ColorList />).find('p').text())
      .toBe('No Colors Listed. (Add a Color)')
  )
})
```

Отображение компонента `Color` без каких-либо свойств охватывает не только строку 8, но и значение свойства по умолчанию, заданное в строке 1 (рис. 10.5).

Мы практически приблизились к 100%-ному охвату тестированием компонента `ColorList`. Единственное, что еще не протестировано, — функции по умолчанию для `onRate` и `onRemove`. Если соответствующие функции не предоставлять, то потребуются эти свойства. Наши показатели тестирования можно улучшить путем отображения компонента `ColorList` без свойств. Кроме того, понадобится имитация щелчков кнопкой мыши на первой кнопке оценки и на последней кнопке удаления:

```
describe("Rendering UI", () => {
  it("Defaults properties correctly", () =>
    expect(shallow(<ColorList />).find('p').text())
      .toBe('No Colors Listed. (Add a Color)')
  )

  it("Clicking default rate button does not cause Error", () => {
    mount(<ColorList colors={_testColors} />)
      .find('button.rate')
      .first()
      .simulate('click')
  })

  it("Clicking default remove button does not cause Error", () => {
    mount(<ColorList colors={_testColors} />)
      .find('button.remove')
      .first()
      .simulate('click')
  })
})
```

При следующем запуске среды Jest с выдачей отчета об охвате станет видно, что теперь тестированием охвачено 100 % кода компонента `ColorList` (рис. 10.6).

Code coverage report for src/ColorList.js.html

All files / src/components/ui/ColorList.js

75% Statements 6/8 60% Branches 3/5 66.67% Functionals 4/6 100% Lines 6/6

```

1 import { PropTypes } from 'react'
2 import Color from './Color'
3 import './../style/sheets/ColorList.scss'
4
5 const ColorList = ({ colors, onRate, onRemove }) => {
6   <div className="color-list">
7     {colors.map((color, i) => {
8       <span key={color.id} style={color.css}> {color.name} </span>
9     })}
10
11     <Color form={color.id}
12       {...onRate}
13       onRate={onRate}
14       onRemove={onRemove} />
15   </div>
16
17   </div>
18
19   ColorList.propTypes = {
20     colors: PropTypes.array,
21     onRate: PropTypes.func,
22     onRemove: PropTypes.func
23   }
24
25   export default ColorList

```

Code coverage generated by Istanbul at Tue Mar 14 2017 13:20:32 GMT-0700 (PDT)

Рис. 10.4. Улучшение показателей охвата за счет тестирования свойства onRemove

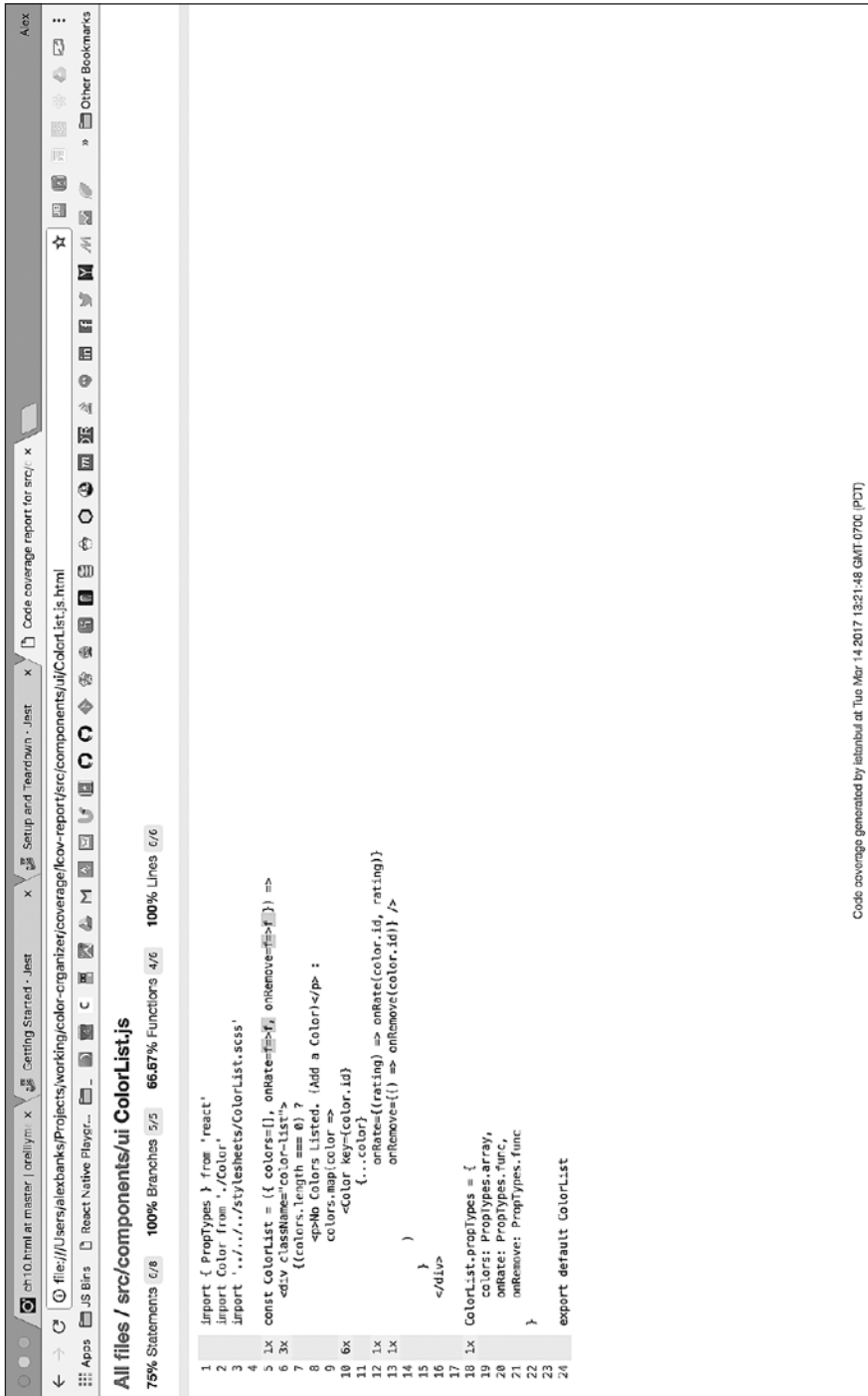


Рис. 10.5. Улучшение показателей охвата за счет тестирования пустого массива цветов

All files / src/components/ui/ColorList.js
 100% Statements 8/8 100% Branches 5/5 100% Functions 6/6 100% Lines 6/6

```

1 import { PropTypes } from 'react'
2 import Color from './Color'
3 import './../styleSheets/ColorList.scss'
4
5 const colorList = ({ colors=[], onRate=>f, onRemove=>rf }) =>
6   {
7     {colors.length === 0} f
8     <div className="color-list">
9       <div> Colors Listed. (Add a color)</div>
10      <div>
11        <Color
12          color={color}
13          onRate={rating} => onRate(color.id, rating)
14          onRemove={() => onRemove(color.id)} />
15      </div>
16    </div>
17
18 ColorList.propTypes = {
19   colors: PropTypes.array,
20   onRate: PropTypes.func,
21   onRemove: PropTypes.func
22 }
23
24 export default ColorList
  
```

Code coverage generated by Istanbul at Tue Mar 14 2017 13:22:38 GMT 0700 (PDT)

Рис. 10.6. Стопроцентный охват тестированием компонента ColorList

Но, как показывают данные на рис. 10.7, остается еще большой объем работы с остальными компонентами нашего проекта. Этим отчетом можно воспользоваться в качестве средства, помогающего понять, как можно повысить качество тестирования за счет увеличения объема кода, охваченного тестированием.

Выбор показателей охвата можно также включить в ваш файл `package.json`:

```
"jest": {
  "setupFiles": ["/__tests__/global.js"],
  "modulePathIgnorePatterns": ["global.js"],
  "moduleNameMapper": {
    "\\.(scss)$": "<rootDir>/node_modules/jest-css-modules"
  },
  "verbose": true,
  "collectCoverage": true,
  "notify": true,
  "collectCoverageFrom": ["src/**"],
  "coverageThreshold": {
    "global": {
      "branches": 80,
      "functions": 80,
      "lines": 80,
      "statements": 80
    }
  }
}
```

Поле `coverageThreshold` определяет объем кода, который должен быть охвачен тестированием, прежде чем тесты будут считаться пройденными. Мы указали, что тестированием нужно охватить 80 % от всех ветвлений, функций, строк кода и инструкций.

В поле `collectCoverageFrom` можно указать, какие файлы должны быть охвачены тестированием. Оно получает массив масок. Мы указали, что тестированием следует охватить все файлы каталога `src` и всех его подкаталогов. Установка для поля `collectCoverage` значения `true` означает, что данные об охвате будут собираться при каждом запуске команды `jest` для этого проекта. Поле `notify` включает вывод окна уведомления с использованием вашей операционной системы. И наконец, в поле `verbose` указывается необходимость вывода подробного отчета о каждом тесте при каждом запуске среды Jest. Подробный отчет для набора `<ColorList /> UI Component` имеет следующий вид:

```
PASS __tests__/components/ui/ColorList.test.js
<ColorList /> UI Component
  Rendering UI
    ✓ Defaults Properties correctly (2ms)
    ✓ Clicking default rate button do not cause Error (6ms)
    ✓ Clicking default remove button do not cause Error (4ms)
  Rating a Color
    ✓ invokes onRate Handler
    ✓ rates the correct color (1ms)
  Removing a Color
    ✓ invokes onRemove Handler
    ✓ removes the correct color
```

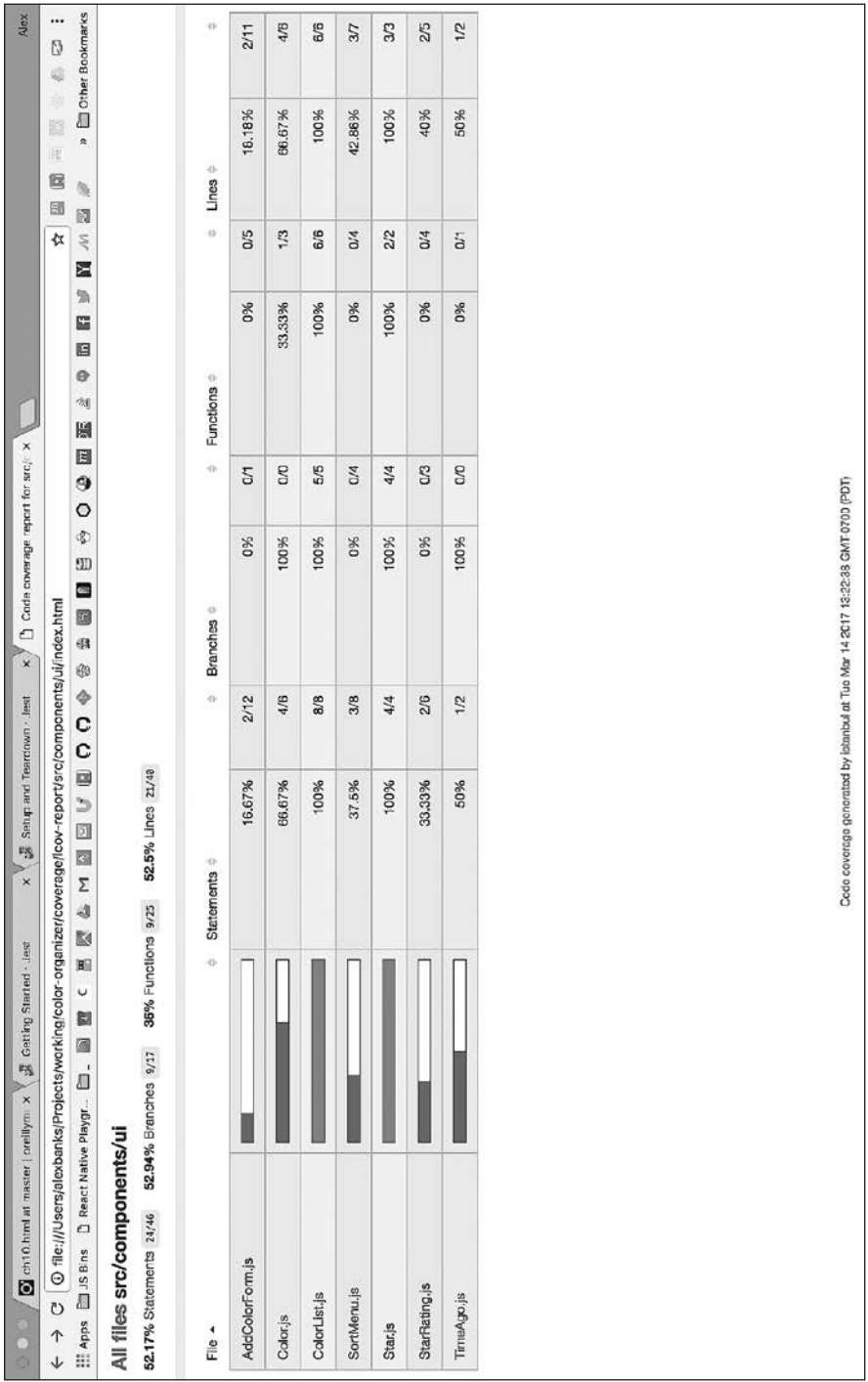


Рис. 10.7. Охват тестированием компонентов пользовательского интерфейса после создания новых тестов для ColorList

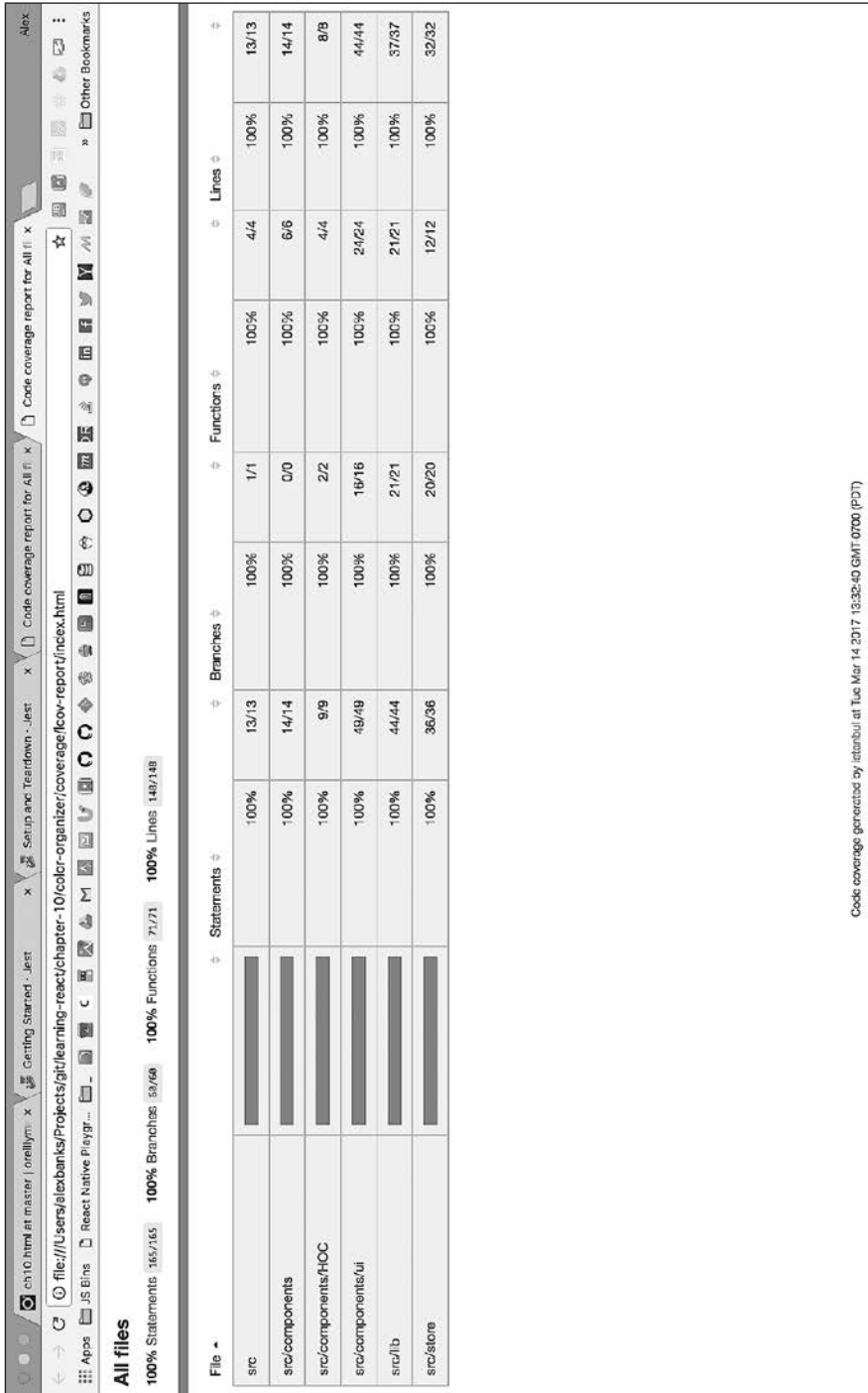


Рис. 10.8. Стопроцентный охват кода тестированием

Чтобы охватить код органайзера цветов тестированием на 100 %, понадобится дополнительный объем тестов. С кодом, который находится в хранилище GitHub для этой главы, достигается именно такой охват кода тестированием (рис. 10.8).

Охват кода тестированием — отличный инструмент, который позволяет оценить глубину тестирования кода. Это один из критериев, помогающих разобраться, является ли достаточным созданный объем блочных тестов для вашего кода. Обычно стремления добиться 100%-ного охвата кода в каждом проекте не наблюдается. Лучше всего будет сориентироваться на 85%-ный охват. Подробнее об этом можно прочитать в статье Мартина Фаулера (Martin Fowler) *Test-Coverage* по адресу <https://martinfowler.com/bliki/TestCoverage.html>.

11

Маршрутизатор React Router

На заре Интернета большинство сайтов состояло из серии страниц, по которым пользователи могли перемещаться путем запросов и открытия отдельных файлов. Местоположение текущего файла или ресурса отображалось в адресной строке браузера. Кнопки перемещения вперед и назад работали вполне ожидаемым образом. Содержимое закладок, нацеленных на глубины сайта, позволяло пользователям сохранять ссылки на конкретный файл, который мог быть заново загружен по запросу пользователя. На сайте со страничной организацией, или с серверным представлением, браузерная навигация и истории просмотров просто работали, как и планировалось.

В одностраничном приложении (single-page application, SPA) описанные функции стали проблематичными. Напоминаем, что в данном приложении все происходит на одной и той же странице. Средства языка JavaScript загружают информацию и вносят изменения в пользовательский интерфейс. Такие свойства, как история просмотров, закладки и кнопки перемещений вперед и назад, не будут работать без решений, связанных с *маршрутизацией*. Она представляет собой процесс определения конечных точек запросов ваших клиентов¹. Эти точки работают в связке с местоположением и объектами истории просмотров браузера. Они используются для идентификации запрошенного содержимого, чтобы средства JavaScript могли загрузить и вывести на экран соответствующий пользовательский интерфейс.

В отличие от Angular, Ember или Backbone, React не поставляется со стандартным маршрутизатором. Осознавая важность решений по маршрутизации, инженеры Майкл Джексон (Michael Jackson) и Райан Флоренс (Ryan Florence) создали маршрутизатор, который назвали просто React Router. Он был благосклонно

¹ Express.js documentation: Basic Routing. <https://expressjs.com/en/starter/basic-routing.html>.

воспринят сообществом в качестве популярного решения по маршрутизации для приложений React¹. Его используют такие компании, как Uber, Zendesk, PayPal и Vimeo².

В этой главе мы представим введение в React Router и обзор приемов использования компонента HashRouter для управления маршрутизацией на стороне клиента.

Встраивание маршрутизатора

Для демонстрации возможностей React Router мы создадим классический стартовый сайт, оснащенный разделами начальной информации — **About**, событий — **Events**, сведений о товарах — **Products** и контактной информации — **Contact Us** (рис. 11.1). Несмотря на то что этот сайт будет смотреться как имеющий несколько страниц, на самом деле страница у него будет только одна, поскольку он являет собой одностраничное приложение (SPA).

[Company Website]



Рис. 11.1. Главная страница сайта компании

Карта этого сайта состоит из главной страницы, страницы для каждого раздела и страницы для обработки ошибок 404 Not Found (Страница не найдена) (рис. 11.2).

<ul style="list-style-type: none"> • Home Page <ul style="list-style-type: none"> – About the Company – Events – Products – Contact Us • 404 Error Page 	<ul style="list-style-type: none"> • http://localhost:3000/ <ul style="list-style-type: none"> – http://localhost:3000/#/about – http://localhost:3000/#/events – http://localhost:3000/#/products – http://localhost:3000/#/contact • http://localhost:3000/#/foo-bar
--	--

Рис. 11.2. Названия страниц и маршруты

¹ Этот отчет был отмечен на GitHub звездочкой более 20 000 раз. <https://github.com/ReactTraining/react-router/stargazers>.

² См. Sites Using React Router. <https://github.com/ReactTraining/react-router/issues/2412>.

Маршрутизатор позволит настроить маршруты для каждого раздела сайта. Каждый *маршрут* является конечной точкой, которая может быть введена в адресную строку браузера. При запросе маршрута можно вывести соответствующее содержимое.



Средство HashRouter

В пакете `react-router-dom` предоставляются два варианта для управления историей просмотров в одностраничных приложениях. Инструмент `HashRouter` был разработан для работы на стороне клиента. Традиционно знаки решетки (`#`) в адресной строке служили для определения точек привязки гипертекстовых ссылок. Когда этот знак применяется в адресной строке, браузер не выполняет запроса к серверу. При использовании `HashRoute` `#` необходимо указывать перед всеми маршрутами.

`HashRouter` хорошо подходит для применения в новых проектах или на небольших клиентских сайтах, не требующих серверного приложения. А `BrowserRouter` является предпочтительным решением для большинства приложений, готовых к использованию на практике. Этот инструмент мы рассмотрим в главе 12, при описании универсальных приложений.

Установим пакет `react-router-dom`, который понадобится для встраивания маршрутизатора в наше приложение на основе браузера:

```
npm install react-router-dom --save
```

Нам также необходимы несколько компонентов-заменителей для каждого раздела или страницы на карте сайта. Их можно экспортировать из одного файла:

```
export const Home = () =>
  <section className="home">
    <h1>[Home Page]</h1>
  </section>

export const About = () =>
  <section className="events">
    <h1>[About the Company]</h1>
  </section>

export const Events = () =>
  <section className="events">
    <h1>[Events Calendar]</h1>
  </section>

export const Products = () =>
  <section className="products">
    <h1>[Products Catalog]</h1>
  </section>

export const Contact = () =>
  <section className="contact">
    <h1>[Contact Us]</h1>
  </section>
```


При запуске приложения вместо отображения одного компонента App мы отобразим компонент HashRouter:

```
import React from 'react'
import { render } from 'react-dom'

import {
  HashRouter,
  Route
} from 'react-router-dom'

import {
  Home,
  About,
  Events,
  Products,
  Contact
} from './pages'

window.React = React

render(
  <HashRouter>
    <div className="main">
      <Route exact path="/" component={Home} />
      <Route path="/about" component={About} />
      <Route path="/events" component={Events} />
      <Route path="/products" component={Products} />
      <Route path="/contact" component={Contact} />
    </div>
  </HashRouter>,
  document.getElementById('react-container')
)
```

Компонент HashRouter отображается в качестве корневого компонента нашего приложения. Каждый маршрут может быть определен внутри HashRouter с помощью компонента Route.

Эти маршруты предписывают маршрутизатору, какой компонент выводить при изменении определения места окна. У каждого компонента Route имеется path и свойства component. Когда определение места в браузере соответствует path, component будет отображен. Если место определено как /, то маршрутизатор выводит компонент Home, а если как /products — то компонент Products.

Первый маршрут, тот, что приводит к выводу компонента Home, обладает свойством exact. Это значит следующее: компонент Home будет выведен только в случае точного соответствия определения места корневному маршруту /.

На данный момент приложение готово к запуску, и мы можем набрать маршрут в адресной строке браузера, чтобы пронаблюдать изменение содержимого. Наберите в адресной строке браузера, к примеру, <http://localhost:3000/#/about>, и пронаблюдайте за выводом компонента About.

Мы не ждем, что пользователи станут перемещаться по сайту, набирая маршруты в адресной строке. Средства, имеющиеся в пакете `react-router-dom`, предоставляют компонент `Link`, который может применяться для создания браузерных ссылок.

Изменим главную страницу таким образом, чтобы она содержала меню навигации со ссылкой для каждого маршрута:

```
import { Link } from 'react-router-dom'

export const Home = () =>
  <div className="home">
    <h1>[Company Website]</h1>
    <nav>
      <Link to="about">[About]</Link>
      <Link to="events">[Events]</Link>
      <Link to="products">[Products]</Link>
      <Link to="contact">[Contact Us]</Link>
    </nav>
  </div>
```

Теперь пользователи могут получить доступ к каждой внутренней странице из главной, щелкнув кнопкой мыши на ссылке. Имеющаяся в браузере кнопка возврата к предыдущей странице вернет их на главную.

Свойства маршрутизатора. React Router передает выводимому им компоненту свойства. Например, через свойство можно получить текущее местоположение. Воспользуемся текущим местоположением `location`, которое поможет создать компонент 404 Not Found:

```
export const Whoops404 = ({ location }) =>
  <div className="whoops-404">
    <h1>Resource not found at '{location.pathname}'</h1>
  </div>
```

Маршрутизатор выведет компонент `Whoops404`, когда пользователи введут маршруты, которые не были определены. После этого маршрутизатор передаст объект `location` этому компоненту в виде свойства. Мы можем взять данный объект и применить для получения текущего путевого имени запрошенного маршрута. С его помощью пользователя уведомят о том, что запрошенный ресурс не может быть найден.

Теперь добавим компонент `Whoops404` к приложению, воспользовавшись `Route`:

```
import {
  HashRouter,
  Route,
  Switch
} from 'react-router-dom'

...

render(
  <HashRouter>
```

```
<div className="main">
  <Switch>
    <Route exact path="/" component={Home} />
    <Route path="/about" component={About} />
    <Route path="/events" component={Events} />
    <Route path="/products" component={Products} />
    <Route path="/contact" component={Contact} />
    <Route component={Whoops404} />
  </Switch>
</div>
</HashRouter>,
document.getElementById('react-container')
)
```

Поскольку компонент `Whoops404` нам нужно будет вывести только в том случае, когда в `Route` не будет никаких других совпадений, понадобится компонент `Switch`. Он выводит только данные первого совпадающего маршрута. Тем самым обеспечивается отображение на экране данных только одного из этих маршрутов. Если ни одному из указанных мест в `Route` не находится совпадений, то будут выведены данные последнего маршрута, то есть того, который не содержит свойства `path`. Допустим, при введении маршрута `http://localhost:3000/#/profits` будет выведено изображение, показанное на рис. 11.3.



Рис. 11.3. Страница ошибки 404

В этом разделе было предоставлено введение в основы реализации и работы с React Router. Маршрутизатор должен охватить все компоненты `Route`, в данном случае `HashRouter`, выбирающий компонент для отображения, основываясь на текущем местоположении окна. Для обеспечения навигации могут использоваться компоненты `Link`. Можно углубляться в основы и дальше, но пока все изложенное касалось лишь поверхностных возможностей маршрутизации.

Вложенные маршруты

Компоненты `Route` применяются с содержимым, которое должно быть выведено только при совпадении с конкретными URL. Эта особенность позволяет упорядочить наши веб-приложения, выстраивая весьма выразительные иерархии, способствующие повторному использованию содержимого.

В данном разделе будут также рассмотрены способы упорядочения содержимого в подразделы, содержащие подпункты меню.

Использование страничного шаблона

Иногда при переходах пользователей по разделам нашего приложения нужно, чтобы часть UI оставалась на месте. В прошлом в деле повторного применения элементов пользовательского интерфейса веб-разработчикам помогали такие решения, как шаблоны страниц и мастер-страницы. Компоненты React могут быть сведены в шаблоны вполне естественным образом с помощью свойства `children`.

Рассмотрим простой стартовый сайт. В нем каждый раздел должен отображать одно и то же главное меню. Содержимое правой части экрана должно меняться по мере переходов пользователя по сайту, в отличие от содержимого левой (рис. 11.4).

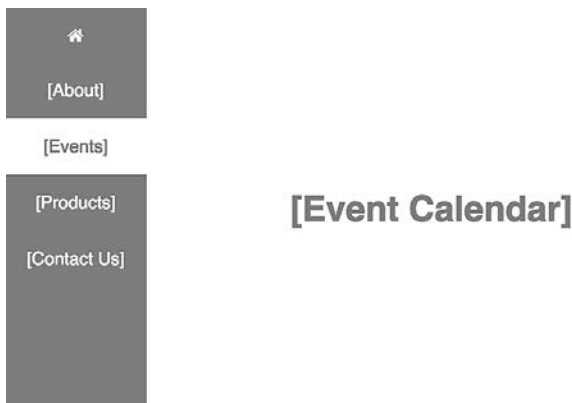


Рис. 11.4. Внутренняя страница событий: Events

Создадим многократно используемый компонент `PageTemplate`, который можно задействовать в качестве шаблона для внутренних страниц. Этот компонент всегда будет отображать главное меню, а кроме того, по мере переходов пользователей по сайту станет выводить на экран вложенное содержимое.

Сначала нам понадобится компонент `MainMenu`:

```
import HomeIcon from 'react-icons/lib/fa/home'
import { NavLink } from 'react-router-dom'
```

```
import './stylesheets/menus.scss'

const selectedStyle = {
  backgroundColor: "white",
  color: "slategray"
}

export const MainMenu = () =>
  <nav className="main-menu">
    <NavLink to="/">
      <HomeIcon/>
    </NavLink>
    <NavLink to="/about" activeStyle={selectedStyle}>
      [About]
    </NavLink>
    <NavLink to="/events" activeStyle={selectedStyle}>
      [Events]
    </NavLink>
    <NavLink to="/products" activeStyle={selectedStyle}>
      [Products]
    </NavLink>
    <NavLink to="/contact" activeStyle={selectedStyle}>
      [Contact Us]
    </NavLink>
  </nav>
```

В компоненте `MainMenu` применяется компонент `NavLink`. Он может использоваться для создания ссылок, которые в случае активности могут получать особое стилевое оформление. Для настройки CSS с целью показа, какая из ссылок является активной или выбранной в данный момент, можно задействовать свойство `activeStyle`.

Компонент `MainMenu` будет использоваться в компоненте `PageTemplate`:

```
import { MainMenu } from './ui/menus'

...

const PageTemplate = ({children}) =>
  <div className="page">
    <MainMenu />
    {children}
  </div>
```

Каждый раздел будет отображаться в дочернем компоненте `PageTemplate`. Здесь мы добавляем дочерний компонент сразу же после `MainMenu`. Теперь можно составить комбинацию из наших разделов, используя `PageTemplate`:

```
export const Events = () =>
  <PageTemplate>
    <section className="events">
      <h1>[Event Calendar]</h1>
    </section>
  </PageTemplate>
```

```

export const Products = () =>
  <PageTemplate>
    <section className="products">
      <h1>[Product Catalog]</h1>
    </section>
  </PageTemplate>

export const Contact = () =>
  <PageTemplate>
    <section className="contact">
      <h1>[Contact Us]</h1>
    </section>
  </PageTemplate>

export const About = ({ match }) =>
  <PageTemplate>
    <section className="about">
      <h1>About</h1>
    </section>
  </PageTemplate>

```

Если запустить приложение, то можно увидеть, что теперь каждый раздел отображает одно и то же главное меню `MainMenu`. Содержимое правой стороны экрана меняется при переходах по внутренним страницам сайта.

Подразделы и подменю

Далее мы вложим в раздел `About` четыре компонента, для чего воспользуемся компонентом `Route` (рис. 11.5).

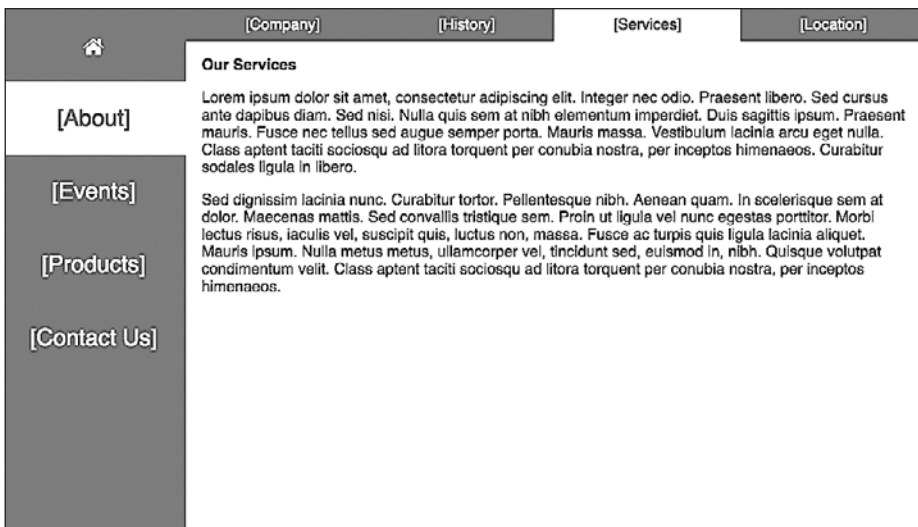


Рис. 11.5. Подстраницы в разделе About

Нам нужно добавить страницы для сведений о компании — `Company`, о ее истории — `History`, об имеющихся службах — `Services` и о ее местонахождении — `Location`. Если пользователь выберет раздел `About`, то изначально должен в этом разделе попадать на страницу `Company`. Схема выглядит следующим образом:

- ❑ Home Page:
 - **About the Company:**
 - **Company (default);**
 - **History;**
 - **Services;**
 - **Location;**
 - Events;
 - Products;
 - Contact Us;
- ❑ 404 Error Page.

Эта иерархия должна отображаться в новых маршрутах, которые нужно будет создать:

- ❑ `http://localhost:3000/;`
 - `http://localhost:3000/#/about:`
 - `http://localhost:3000/#/about;`
 - `http://localhost:3000/#/about/history;`
 - `http://localhost:3000/#/about/services;`
 - `http://localhost:3000/#/about/location;`
 - `http://localhost:3000/#/events;`
 - `http://localhost:3000/#/products;`
 - `http://localhost:3000/#/contact;`
- ❑ `http://localhost:3000/#/foo-bar.`

Создадим подменю для раздела `About`. Для этого воспользуемся компонентами `NavLink` и настроим свойство `activeStyle` точно так же, как и свойство `activeStyle`, применяемое в `MainMenu`:

```
export const AboutMenu = ({match}) =>
  <div className="about-menu">
    <li>
      <NavLink to="/about"
        style={match.isExact && selectedStyle}>
```

```

        [Company]
      </NavLink>
    </li>
    <li>
      <NavLink to="/about/history"
        activeStyle={selectedStyle}>
        [History]
      </NavLink>
    </li>
    <li>
      <NavLink to="/about/services"
        activeStyle={selectedStyle}>
        [Services]
      </NavLink>
    </li>
    <li>
      <NavLink to="/about/location"
        activeStyle={selectedStyle}>
        [Location]
      </NavLink>
    </li>
  </div>

```

Чтобы направить пользователей на внутреннее содержимое раздела **About**, компоненты `NavLink` применяются в компоненте `AboutMenu`. Он будет отображаться с помощью `Route`, стало быть, получает свойства маршрутизации. Нам придется задействовать свойство `match`, отправляемое этому компоненту из `Route`.

Во всех компонентах `NavLink`, кроме самого первого, применяется свойство `activeStyle`. Оно будет задавать свойство стиля для ссылки, когда местоположение совпадет с маршрутом ссылки. Например, в случае перехода пользователя на `http://localhost:3000/about/services` в `Services` компонент `NavLink` будет выведен на белом фоне.

В первом компоненте `NavLink` свойство `activeStyle` не используется. Вместе с тем свойство стиля назначается `selectedStyle`, только если маршрут в точности соответствует `/about`. Свойство точного соответствия `match.isExact` получает значение `true`, когда местоположение соответствует маршруту `/about`, а значение `false` оно получает, когда местоположение соответствует маршруту `/about/services`. С технической точки зрения маршрут `/about` соответствует обоим местам, но точное соответствие получается, только если местоположение соответствует маршруту `/about`.



Компоненты-заместители

Нужно также не забыть заглушить компонентами-заместителями наши новые разделы: `Company`, `Services`, `History` и `Location`. Вот пример заместителя компонента `Services`. Он просто выводит некий замещающий текст:


```

export const Services = () =>
  <section className="services">
    <h2>Our Services</h2>
    <p>
      Lorem ipsum dolor sit amet, consectetur
      adipiscing elit. Integer nec odio.
      Praesent libero. Sed cursus ante dapibus
      diam. Sed nisi. Nulla quis sem at nibh
      elementum imperdiet. Duis sagittis ipsum.
      Praesent mauris. Fusce nec tellus sed
      augue semper porta. Mauris massa.
      Vestibulum lacinia arcu eget nulla.
      Class aptent taciti sociosqu ad litora
      torquent per conubia nostra, per inceptos
      himenaeos. Curabitur sodales ligula in
      libero.
    </p>
    <p>
      Sed dignissim lacinia nunc. Curabitur
      tortor. Pellentesque nibh. Aenean quam. In
      scelerisque sem at dolor. Maecenas mattis.
      Sed convallis tristique sem. Proin ut
      ligula vel nunc egestas porttitor. Morbi
      lectus risus, iaculis vel, suscipit quis,
      luctus non, massa. Fusce ac turpis quis
      ligula lacinia aliquet. Mauris ipsum.
      Nulla metus metus, ullamcorper vel,
      tincidunt sed, euismod in, nibh. Quisque
      volutpat condimentum velit. Class patent
      taciti sociosqu ad litora torquent per
      conubia nostra, per inceptos himenaeos.
    </p>
  </section>

```

Теперь мы готовы добавить маршруты к компоненту **About**:

```

export const About = ({ match }) =>
  <PageTemplate>
    <section className="about">
      <Route component={AboutMenu} />
      <Route exact path="/about" component={Company}/>
      <Route path="/about/history" component={History}/>
      <Route path="/about/services" component={Services}/>
      <Route path="/about/location" component={Location}/>
    </section>
  </PageTemplate>

```

Этот компонент **About** будет многократно использоваться по всему разделу. Местоположение сообщит приложению, какой именно подраздел нужно вывести на экран. Например, когда местоположение соответствует адресу <http://localhost:300/about/history>, внутри компонента **About** окажется выведен компонент **History**.

На этот раз мы не используем компонент `Switch`. Любой компонент `Route`, соответствующий местоположению, выведет на экран связанный с ним компонент. Первый компонент `Route` всегда будет показывать на экране `AboutMenu`. Кроме того, выводить связанные с ним компоненты станут и любые другие компоненты `Route` с найденным соответствием.

Использование перенаправлений. Иногда возникает потребность перенаправить пользователей с одного маршрута на другой. Например, мы можем убедиться, что если посетители сайта пытаются получить доступ к содержимому на `http://localhost:3000/services`, то они перенаправляются по правильному маршруту: `http://localhost:3000/about/services`.

Внесем изменения в наше приложение, включив в него перенаправления, гарантирующие возможность доступа пользователей к правильному содержимому:

```
import {
  HashRouter,
  Route,
  Switch,
  Redirect
} from 'react-router-dom'

...

render(
  <HashRouter>
    <div className="main">
      <Switch>
        <Route exact path="/" component={Home} />
        <Route path="/about" component={About} />
        <Redirect from="/history" to="/about/history" />
        <Redirect from="/services" to="/about/services" />
        <Redirect from="/location" to="/about/location" />
        <Route path="/events" component={Events} />
        <Route path="/products" component={Products} />
        <Route path="/contact" component={Contact} />
        <Route component={Whoops404} />
      </Switch>
    </div>
  </HashRouter>,
  document.getElementById('react-container')
)
```

Компонент `Redirect` позволяет перенаправить пользователя по заданному маршруту.



Когда маршруты меняются в рабочем приложении, пользователи могут по старой памяти пытаться получить доступ по прежним маршрутам. Обычно такое происходит из-за применения закладок. Компонент `Redirect` предоставляет способ загрузить правильное содержимое для пользователей, даже если они обращаются к вашему сайту с помощью устаревших закладок.

React Router позволяет составлять композиции из компонентов `Route` в любом месте приложения, поскольку нашим корневым компонентом является `HashRouter`. Теперь можно выстроить содержимое в иерархии, облегчающие переходы от одних данных к другим.

Параметры маршрутизатора

Еще одна полезная возможность React Router — настройка *параметров маршрутизации*. Они представляют собой переменные, получающие свои значения из URL. Эти параметры особенно полезны в веб-приложениях, управляемых данными, где применяются для фильтрации содержимого или управления настройками отображения данных.

Добавление страницы с информацией о цвете

Усовершенствуем органайзер цветов, добавив возможность выбора и отображения какого-нибудь одного цвета с помощью React Router. Когда пользователь выбирает цвет щелчком кнопкой мыши, приложение должно его отобразить, показав `title` и значение `hex` (рис. 11.6).



Рис. 11.6. Страница с информацией о цвете

У каждого цвета имеется уникальный идентификатор. Он может использоваться для поиска конкретных цветов, сохраненных в состоянии. Можно, к примеру, создать функцию `findById`, выполняющую поиск объекта в массиве по его полю `id`:

```
import { compose } from 'redux'

export const getFirstArrayItem = array => array[0]

export const filterArrayById = (array, id) =>
  array.filter(item => item.id === id)
```

```
export const findById = compose(
  getFirstArrayItem,
  filterArrayById
)
```

Эта функция `findById` следует методам функционального программирования, рассмотренным в главе 2. Можно заметить, что метод `findById` сначала фильтрует массив, используя идентификатор, а затем возвращает первый элемент, найденный в отфильтрованном массиве. Функция `findById` пригодна для поиска цветов в состоянии по их уникальным идентификаторам.

С помощью маршрутизатора можно получить идентификатор цвета через URL. Например, в результате того, что идентификатор для цвета `lawn` был передан в URL, этот адрес, используемый для отображения цвета `Lawn`, приобрел следующий вид:

```
http://localhost:3000/#/58d9caee-6ea6-4d7b-9984-65b145031979
```

Параметры маршрутизатора позволяют изъять данное значение. Эти параметры могут быть определены в маршрутах с помощью точки с запятой. Например, уникальный идентификатор можно получить и сохранить в параметре `id`, задействуя компонент `Route`:

```
<Route exact path="/:id" component={UniqueIDHeader} />
```

Компонент `UniqueIDHeader` может получить идентификатор из объекта `match.params`:

```
const UniqueIDHeader = ({ match }) => <h1>{match.params.id}</h1>
```

Параметры можно создавать всякий раз, когда необходимо собирать данные из URL.

Работа с несколькими параметрами

Создать несколько параметров и получить к ним доступ можно с помощью одного объекта параметров. Следующий образец маршрута создает три параметра:

```
<Route path="/members/:gender/:state/:city"
  component="Member" />
```

Эти три параметра могут быть проинициализированы с помощью URL:

```
http://localhost:3000/members/female/california/truckee
```

Все три значения будут переданы компоненту `Member` путем использования `match.params`:

```
const Member = ({ match }) =>
  <div className="member">
    <ul>
      <li>gender: {match.params.gender}</li>
      <li>state: {match.params.state}</li>
      <li>city: {match.params.city}</li>
    </ul>
  </div>
```

Создадим компонент `ColorDetails`, который будет отображаться на экране, когда пользователь выбирает один цвет:

```
const ColorDetails = ({ title, color }) =>
  <div className="color-details"
    style={{backgroundColor: color}}>
    <h1>{title}</h1>
    <h1>{color}</h1>
  </div>
```

Компонент `ColorDetails` относится к презентационным компонентам — он ожидает свойств, содержащих информацию о цвете. Поскольку мы применяем `Redux`, нужно будет добавить новый контейнер, способный найти выбранный цвет в состоянии, используя параметр маршрутизации:

```
export const Color = connect(
  (state, props) => findById(state.colors, props.match.params.id)
)(ColorDetails)
```

Контейнер `Color` создан с помощью компонента высшего порядка `connect`. Первый аргумент является функцией, используемой для установки свойств `ColorDetails` на основе данных одного цвета, взятого из состояния. Задействуя функцию `findById`, которую мы определили ранее в данном разделе, найдем в состоянии отдельный объект цвета с параметром идентификатора `id`, полученным из URL. НОС `connect` отобразит данные из найденного объекта цвета на свойства компонента `ColorDetails`.

НОС `connect` также отобразит на компонент `ColorDetails` любые свойства, отправленные контейнеру `Color`. Это значит, что компоненту `ColorDetails` будут переданы и все свойства маршрутизатора.

Воспользуемся свойством маршрутизатора `history` и добавим к компоненту `ColorDetails` переходы:

```
const ColorDetails = ({ title, color, history }) =>
  <div className="color-details"
    style={{backgroundColor: color}}
    onClick={() => history.goBack()}>
    <h1>{title}</h1>
    <h1>{color}</h1>
  </div>
```

Когда пользователь щелкнет кнопкой мыши на элементе `div.color-details`, будет вызван метод `history.goBack()` и пользователь вернется к предыдущему месту просмотра.

Теперь, имея в своем распоряжении контейнер `Color`, нам нужно добавить его к приложению. Сначала следует поместить компонент `App` при его начальном отображении в оболочку `HashRouter`:

```
import { HashRouter } from 'react-router-dom'
...
render(
  <Provider store={store}>
```

```

    <HashRouter>
      <App />
    </HashRouter>
  </Provider>,
  document.getElementById('react-container')
)

```

А теперь мы готовы настроить маршруты в любом месте нашего приложения. Добавим к компоненту `App` несколько маршрутов:

```

import { Route, Switch } from 'react-router-dom'
import Menu from './ui/Menu'
import { Colors, Color, NewColor } from './containers'
import '../stylesheets/APP.scss'

const App = () =>
  <Switch>
    <Route exact path="/:id" component={Color} />
    <Route path="/"
      component={() => (
        <div className="app">
          <Menu />
          <NewColor />
          <Colors />
        </div>
      )} />
  </Switch>

export default App

```

Компонент `Switch` используется для отображения данных по одному из двух маршрутов: компонентов отдельного цвета или главного приложения. Первый компонент `Route` отображает компонент `Color`, когда в URL передается идентификатор. Например, соответствие этому маршруту будет определено, когда местоположение обозначено следующим образом:

```
http://localhost:3000/#/58d9caee-6ea6-4d7b-9984-65b145031979
```

Любые другие указания местоположения будут соответствовать каталогу `/` и приведут к отображению главных компонентов приложения. Второй компонент `Route` группирует несколько компонентов под новым безымянным функциональным компонентом, не имеющим состояния. В результате пользователи, в зависимости от содержимого URL, увидят либо отдельно взятый цвет, либо список цветов.

В данный момент приложение можно протестировать, добавив параметр `id` непосредственно в адресную строку браузера. Но для перехода к просмотру информации о цвете пользователям понадобится соответствующий способ.

На этот раз компонент `NavLink` не будет применяться для управления переходом от списка цветов к просмотру информации об отдельно взятых цветах. Вместо этого переход будет выполняться путем непосредственного использования принадлежащего маршрутизатору объекта `history`.

Добавим переходы к компоненту `Color`, который находится в папке `./ui`. Данный компонент отображается с помощью компонента `ColorList`. Он не получает свойства маршрутизации из компонента `Route`. Эти свойства можно явным образом передать компоненту `Color` в любом месте ниже по дереву, но проще будет воспользоваться функцией `withRouter`. Она поставляется вместе с модулем `react-router-dom`. Функция может применяться для добавления свойств маршрутизации к любому компоненту, который отображается где-то под компонентом `Route`.

С помощью функции `withRouter` можно получить принадлежащий маршрутизатору объект `history` в виде свойства. Ею можно воспользоваться для переходов внутри компонента `Color`:

```
import { withRouter } from 'react-router'

...

class Color extends Component {
  render() {
    const {
      id,
      title,
      color,
      rating,
      timestamp,
      onRemove,
      onRate,
      history } = this.props

    return (
      <section className="color" style={this.style}>
        <h1 ref="title"
          onClick={() => history.push(`/${id}`)}>
          {title}
        </h1>
        <button onClick={onRemove}>
          <FaTrash />
        </button>
        <div className="color"
          onClick={() => history.push(`/${id}`)}
          style={{ backgroundColor: color }}>
        </div>
        <TimeAgo timestamp={timestamp} />
        <div>
          <StarRating starsSelected={rating}
            onRate={onRate}/>
        </div>
      </section>
    )
  }
}

export default withRouter(Color)
```

Функция `withRouter` является НОС. При экспортировании компонента `Color` мы отправляем его этой функции, которая заключает его в компонент, передающий свойства маршрутизатора: `match`, `history` и `location`.

Переходы получаются благодаря непосредственному применению объекта `history`. Когда пользователь щелкает кнопкой мыши на самом названии цвета, в объект `history` внедряется новый маршрут, представляющий собой строку, содержащую идентификатор цвета. Внедрение этого маршрута в историю переходов приведет к выполнению перехода.

Единый источник истины?

На данный момент состояние органайзера цветов обрабатывается в основном хранилищем `Redux`. Кроме того, у нас имеется состояние, обрабатываемое маршрутизатором. При этом, если в маршруте содержится идентификатор цвета, презентационное состояние приложения отличается от того, которое бывает при отсутствии идентификатора.

Казалось бы, наличие состояния, обрабатываемого маршрутизатором, противоречит требованию `Redux` по хранению состояния в одном объекте: в едином источнике истины. Но маршрутизатор можно возвести в ранг источника истины, обеспечивающего согласование с браузером. Вполне допустимо разрешить маршрутизатору обрабатывать любое состояние, связанное с картой сайта, включая фильтры, требуемые для поиска данных. А все остальное состояние следует содержать в хранилище `Redux`.

Перемещение состояния сортировки цветов в маршрутизатор

Ограничивать сферу применения параметров `Router` не обязательно. Они могут быть не только фильтрами для поиска конкретных данных в состоянии. Их также можно применять для получения информации, необходимой для отображения пользовательского интерфейса.

На данный момент хранилище `Redux` содержит в свойстве `sort` информацию о порядке сортировки цветов в состоянии. Но, возможно, целесообразнее будет переместить эту переменную из хранилища `Redux` в параметр маршрута? Сама по себе переменная данными не является, она просто информирует о том, как они должны быть представлены. Переменная `sort` — строка; это также делает ее идеальным кандидатом для параметра маршрута. И наконец, нам нужно, чтобы пользователи могли отправлять в ссылке состояние сортировки другим пользователям. Если они предпочитают сортировать цвета по рейтингу, то могут отправить эту информацию другим пользователям по ссылке или же добавить закладку в браузере.

Переместим состояние сортировки отображения цветов в параметр маршрута. Используемые для сортировки цветов маршруты имеют следующий вид:

- ❑ `/#/ default` — сортировка по дате;
- ❑ `/#/sort/title` — сортировка по названию;
- ❑ `/#/sort/rating` — сортировка по рейтингу.

Сначала нужно удалить преобразователь сортировки из файла `./store/index.js`, поскольку он больше не понадобится. В результате из кода:

```
combineReducers({colors, sort})
```

получится код:

```
combineReducers({colors})
```

Удаление преобразователя означает, что Redux больше не будет обрабатывать переменную состояния.

Затем можно удалить из файла `./src/components/containers.js` контейнер для компонента `Menu`. Данный контейнер используется для связи состояния хранилища Redux с презентационным компонентом `Menu`. Сортировка уже не хранится в состоянии, поэтому контейнер больше не нужен.

Кроме того, следует внести изменения в контейнер `Colors` в файле `containers.js`. Он больше не будет получать из состояния значение сортировки. Вместо этого он станет получать инструкции по сортировке в качестве параметра маршрута, передаваемого компоненту `Color` внутри свойства `match`:

```
export const Colors = connect(
  ({colors}, {match}) =>
    ({
      colors: sortColors(colors, match.params.sort)
    }),
  dispatch =>
    ({
      onRemove(id) {
        dispatch(removeColor(id))
      },
      onRate(id, rating) {
        dispatch(rateColor(id, rating))
      }
    })
)(ColorList)
```

Теперь перед передачей в виде свойства компоненту `ColorList` цвета будут сортироваться через параметр маршрутизации.

После этого понадобится заменить компонент `Menu` другим компонентом, содержащим ссылки на новые маршруты. Во многом так же, как и в меню `About`, созданном ранее в данной главе, визуальное состояние меню будет управляться путем установки свойства `activeStyle`, принадлежащего компоненту `NavLink`:

```
import { NavLink } from 'react-router'

const selectedStyle = { color: 'red' }
```

```
const Menu = ({ match }) =>
  <nav className="menu">
    <NavLink to="/" style={match.isExact && selectedStyle}>
      date
    </NavLink>
    <NavLink to="/sort/title" activeStyle={selectedStyle}>
      title
    </NavLink>
    <NavLink to="/sort/rating" activeStyle={selectedStyle}>
      rating
    </NavLink>
  </nav>
```

```
export default Menu
```

Теперь пользователи могут сортировать цвета с помощью URL. При недоступности параметра сортировки цвета будут сгруппированы по датам. Чтобы показать пользователю способ, примененный для сортировки данных, это меню будет изменять цвет ссылки.

Нужно внести изменения в компонент `App`, чтобы обработка сортировки цветов велась через маршруты:

```
const App = () =>
  <Switch>
    <Route exact path="/:id" component={Color} />
    <Route path="/" component={() => (
      <div className="app">
        <Route component={Menu} />
        <NewColor />
        <Switch>
          <Route exact path="/" component={Colors} />
          <Route path="/sort/:sort" component={Colors} />
        </Switch>
      </div>
    )} />
  </Switch>
```

Сначала для `Menu` понадобится свойство соответствия, следовательно, отображаться `Menu` будет с помощью `Route`. Компонент `Menu` всегда будет отображаться рядом с формой `NewColor` и списком цветов, поскольку у `Route` нет пути.

После компонента `NewColor` нужно вывести либо исходный список цветов, отсортированный по умолчанию, либо список, отфильтрованный по параметру. Эти маршруты заключены в компонент `Switch`, чтобы обеспечить отображение только одного контейнера `Colors`.

Когда пользователи переходят к маршруту главной страницы, `http://localhost:3000`, отображается компонент `App`. По умолчанию внутри `App` отображается контейнер `Colors`. Параметр сортировки имеет неопределенное значение (`undefined`), поэтому цвета сортируются по умолчанию.

При переходе пользователя на `http://localhost:3000/sort/rating` также отображается контейнер `Colors`, но теперь параметр сортировки будет иметь конкретное значение, и цвета должны быть отсортированы в соответствии с этим значением.

Параметры маршрутизации являются идеальным инструментом для получения данных, влияющих на презентацию вашего UI. Но они должны применяться только когда нужно, чтобы пользователь получал соответствующие подробности через URL. Например, в случае с органайзером цветов пользователи могут отправлять другим пользователям ссылки на конкретные цвета или на все образцы, отсортированные по указанному полю. Пользователи могут также создавать закладки для этих ссылок с целью вернуть конкретные данные. Если нужно, чтобы ваши пользователи сохраняли информацию о презентации в URL, то самым подходящим решением станет применение параметров маршрутизации.

В данной главе мы показали основные приемы использования утилиты `React Router`. Во всех примерах главы применялся компонент `HashRouter`. В следующей главе мы продолжим задействовать маршрутизатор на стороне как клиента, так и сервера, воспользовавшись компонентом `BrowserRouter`, а для отображения текущего контекста маршрутизации на сервере послужит компонент `StaticRouter`.

12 React и сервер

До сих пор мы создавали небольшие приложения с React, запускаемые исключительно в браузере. Они собирали данные в браузере, а те содержались в его хранилище. В этом был вполне определенный смысл, поскольку React относится к уровню представления данных. Его предназначением является отображение пользовательского интерфейса. И тем не менее для многих приложений требуется как минимум какая-либо серверная часть программы, а нам нужно понимать, как создавать структуру приложений с учетом применения сервера.

Даже если ваше клиентское приложение целиком зависит по серверной части от облачных сервисов, вам все равно понадобится получать и отправлять данные, используя эти сервисы. В архитектуре Flux имеются конкретные места, где должны совершаться такие транзакции, а также библиотеки, способные справиться с задержками, связанными с HTTP-запросами.

Кроме того, React способна на *изоморфные* режимы отображения, то есть может применяться на платформах, отличных от браузера. Следовательно, пользовательский интерфейс можно отображать на сервере перед тем, как он попадет в браузер.

Применяя серверное отображение, можно повысить производительность, переносимость и безопасность ваших приложений.

Данную главу мы начнем с просмотра различий между изоморфизмом и универсализмом и того, какое отношение обе эти концепции имеют к React. Затем рассмотрим способы создания изоморфных приложений с помощью универсального кода JavaScript. И наконец, усовершенствуем органайзер цветов, добавив к нему сервер, а также первоначальное отображение пользовательского интерфейса на сервере.

Сравнение изоморфизма с универсализмом

Термины «*изоморфный*» и «*универсальный*» часто применяются для описания приложений, работающих как на клиентской, так и на серверной стороне. Хотя для описания одного и того же приложения эти термины используются, заменяя друг друга, между ними все же есть тонкие различия. *Изоморфными* считаются те приложения, которые могут отображаться на нескольких платформах. А понятие *универсального* кода означает, что абсолютно одинаковый код может запускаться в нескольких средах¹.

Среда Node.js позволит многократно задействовать один и тот же код, созданный в браузере, в других приложениях, таких как серверы, интерфейсы командной строки и даже автономные приложения. Посмотрим на фрагмент универсального кода JavaScript:

```
var printNames = response => {
  var people = JSON.parse(response).results,
      names = people.map(({name}) => `${name.last}, ${name.first}`)
  console.log(names.join('\n'))
}
```

Функция `printNames` имеет универсальный характер. Тот же самый код можно вызвать и в браузере, и на сервере. Это значит, что при создании сервера с помощью среды Node.js потенциально возможно повторно использовать большие объемы кода между двумя средами. Универсальный код JavaScript относится к такому коду JavaScript, который может запускаться на сервере или в браузере, не выдавая ошибок (рис. 12.1).

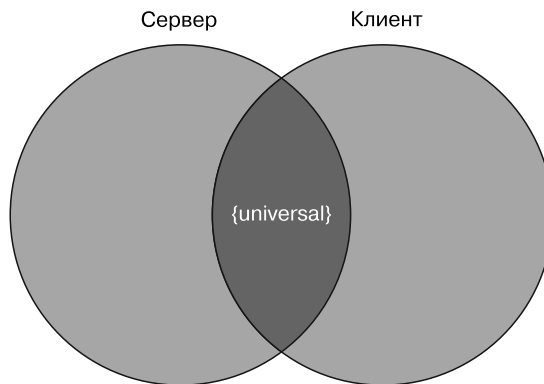


Рис. 12.1. Области клиентской и серверной сторон

¹ Hengeveld G. Isomorphism vs Universal JavaScript. <https://medium.com/@ghengeveld/isomorphism-vs-universal-javascript-4b47fb481beb>, Medium.

Серверная и клиентская области имеют совершенно разную природу, следовательно, весь ваш код JavaScript не будет автоматически работать в обеих средах. Посмотрим на создание Ajax-запроса с помощью браузера:

```
const request = new XMLHttpRequest()
request.open('GET', 'https://api.randomuser.me/?nat=US&results=10')
request.onload = () => printNames(request.response)
request.send()
```

Здесь запрашиваются десять произвольных пользовательских записей из API `randomuser.me`. Если запустить этот код в браузере и посмотреть на консоль, станут видны десять произвольно взятых имен:

```
ford, brianna
henderson, nellie
lynch, lily
gordon, todd
collins, genesis
roberts, suzanne
dixon, rene
ray, rafael
adams, jamie
bowman, mia
```

Но при попытке запустить этот же код в среде Node.js будет выдана ошибка:

```
ReferenceError: XMLHttpRequest is not defined
    at Object.<anonymous> (/Users/...)
    at Module._compile (module.js:541:32)
    at Object.Module._extensions..js (module.js:550:10)
    at Module.load (module.js:458:32)
    at tryModuleLoad (module.js:417:12)
    at Function.Module._load (module.js:409:3)
    at Function.Module.runMain (module.js:575:10)
    at startup (node.js:160:18)
    at node.js:449:3
```

Дело в том, что в среде Node.js нет объекта `XMLHttpRequest`, который есть в браузере. В Node.js для выполнения запроса можно воспользоваться модулем `http`:

```
const https = require('https')
https.get(
  'https://api.randomuser.me/?nat=US&results=10',
  res => {
    let results = ''

    res.setEncoding('utf8')
    res.on('data', chunk => results += chunk)

    res.on('end', () => printNames(results))
  }
)
```

Загрузка данных из API в среде Node.js требует использования основных модулей. Здесь нужен другой код. В показанных примерах функция `printNames` имеет универсальный характер, поэтому работает в обеих средах.

Можно создать модуль, который будет выводить имена в консоль либо в браузере, либо в приложении Node.js:

```
var printNames = response => {
  var people = JSON.parse(response).results,
  names = people.map(({name}) => `${name.last}, ${name.first}`)
  console.log(names.join('\n'))
}

if (typeof window !== 'undefined') {
  const request = new XMLHttpRequest()
  request.open('GET', 'http://api.randomuser.me/?nat=US&results=10')
  request.onload = () => printNames(request.response)
  request.send()
} else {
  const https = require('https')
  https.get(
    'http://api.randomuser.me/?nat=US&results=10',
    res => {
      let results = ''
      res.setEncoding('utf8')
      res.on('data', chunk => results += chunk)
      res.on('end', () => printNames(results))
    }
  )
}
```

Теперь этот файл JavaScript имеет изоморфный характер, поскольку в нем содержится универсальный код JavaScript. Весь код нельзя назвать универсальным, но сам файл будет работать в обеих средах. Его можно запустить с помощью среды Node.js или же включить в тег `<script>` в браузере.



Функция `isomorphic-fetch`

При извлечении данных среди всех других WHATWG-реализаций функции извлечения данных мы отдали предпочтение функции `isomorphic-fetch`, поскольку она работает в нескольких средах.

Посмотрим на компонент `Star`. Можно ли считать его универсальным?

```
Const Star = ({ selected=false, onClick=f=>f }) =>
  <div className={selected ? "star selected" : "star"}
    onClick={onClick}>
  </div>
```

Конечно: вспомним, что код JSX компилируется в код JavaScript. Компонент `Star` — это просто функция:

```
const Star = ({ selected=false, onClick=f=>f }) =>
  React.createElement(
    "div",
    {
      className: selected ? "star selected" : "star",
      onClick: onClick
    }
  )
```

Этот компонент можно отобразить непосредственно в браузере или же в другой среде и забрать HTML-вывод в виде строки. В библиотеке ReactDOM есть метод `renderToString`, который можно применить для отображения пользовательского интерфейса в строке кода HTML:

```
// Отображение HTML-кода непосредственно в браузере
ReactDOM.render(<Star />)
```

```
// Отображение HTML-кода в виде строки
var html = ReactDOM.renderToString(<Star />)
```

Можно создавать изоморфные приложения, отображающие компоненты на различных платформах и выстраивать архитектуру этих приложений так, чтобы код JavaScript использовался повторно универсальным образом во множестве сред. Кроме того, можно создавать изоморфные приложения, задействуя другие языки программирования, такие как Go или Python. Мы не ограничены одной лишь средой Node.js.

Код React, отображаемый на сервере

Метод `ReactDOM.renderToString` дает возможность отображать пользовательский интерфейс на сервере. Серверы обладают большими вычислительными возможностями, у них есть доступ к разнообразным ресурсам, которых нет у браузеров. Серверы могут быть безопасными и иметь доступ к безопасным данным. Все эти дополнительные преимущества можно применять, отобразив исходное содержимое на сервере.

Создадим базовый веб-сервер, воспользовавшись Node.js и Express. Библиотеку Express мы задействуем как средство быстрой разработки веб-серверов:

```
npm install express -save
```

Посмотрим на простое Express-приложение. Данный код создает веб-сервер, который всегда выдает сообщение `Hello World`. Сначала информация о каждом запросе регистрируется в консоли. Затем сервер выдает ответ в виде кода HTML. Оба шага содержатся в своих собственных функциях и объединяются в цепочку

с помощью метода `.use()`. Express автоматически вставляет аргументы запроса и ответа в каждую из этих функций.

```
import express from 'express'

const logger = (req, res, next) => {
  console.log(`${req.method} request for '${req.url}'`)
  next()
}

const sayHello = (req, res) =>
  res.status(200).send("<h1>Hello World</h1>")

const app = express()
  .use(logger)
  .use(sayHello)

app.listen(3000, () =>
  console.log(`Recipe app running at 'http://localhost:3000'`)
)
```

Связующим кодом являются функции `logger` и `sayHello`. В Express функции связующего кода выстраиваются друг за другом в конвейер, для чего используется метод `.use()`¹. При выдаче запроса каждая связующая функция вызывается в порядке следования в цепочке конвейера, пока не будет отправлен ответ. Это приложение Express регистрирует подробности каждого запроса в консоли, а затем отправляет HTML-ответ: `<h1>Hello World</h1>`. И наконец, Express-приложение запускается путем выдачи ему указания о локальном отслеживании поступающих запросов на порте 3000.

В главе 10 для запуска наших тестов мы использовали пакет `babel-cli`. Здесь мы используем его, чтобы запустить это Express-приложение, потому что в нем содержится инструкция ES6-синтаксиса `import`, не поддерживаемая текущей версией среды Node.js.



Использование `babel-cli` не считается удачным решением для запуска приложений в режиме их практического применения, так что не нужно применять `babel-cli` для запуска каждого приложения Node.js, в котором задействован синтаксис спецификации ES6. Когда шла работа над этой книгой, текущая версия Node.js уже поддерживала большой объем синтаксиса ES6. Можно просто отказаться от применения инструкций `import`, которые будут поддерживаться будущими версиями Node.js.

Еще один вариант заключается в создании сборки Webpack для вашего кода, работающего на стороне сервера. Утилита Webpack может экспортировать пакет JavaScript, который способен запускаться с устаревшими версиями Node.js.

¹ Документация Express: Using Middleware. <https://expressjs.com/en/guide/using-middleware.html>.

Чтобы запустить `babel-node`, нужно выполнить ряд настроек. Сначала необходимо установить `babel-cli`, `babel-loader`, `babel-preset-es2015`, `babel-preset-react` и `babel-preset-stage-0`:

```
npm install babel-cli babel-loader babel-preset-env
babel-preset-react babel-preset-stage-0 -save
```

Затем следует убедиться, что файл `.babelrc` добавлен в корневой каталог нашего проекта. При запуске команды `babel-node index-server.js` Babel станет искать этот файл и применять установленные нами предварительные настройки:

```
{
  "presets": [
    "env",
    "stage-0",
    "react"
  ]
}
```

И наконец, добавим к нашему файлу `package.json` сценарий `start`. Если данного файла еще не существует, то его нужно создать, запустив команду `npm init`:

```
"scripts": {
  "start": "./node_modules/.bin/babel-node index-server.js"
}
```

Теперь можно запустить наш Express-сервер, воспользовавшись командой `npm start`:

```
npm start
```

```
Recipe app running at 'http://localhost:3000'
```

После запуска сервера можно открыть браузер и перейти на `http://localhost:3000`. На странице будет показано сообщение Hello World.



Остановить работу Express-сервера можно нажатием сочетания клавиш `Ctrl+C`.

До сих пор Express-приложение отзывалось на все запросы одной и той же строкой: `<h1>Hello world</h1>`. Вместо вывода этого сообщения отобразим приложение `Recipes`, с которым мы работали в главах 4 и 5. Соответствующие изменения можно внести путем отображения компонента `Menu` с данными рецепта, воспользовавшись методом `renderToString` из `ReactDOM`:

```
import React from 'react'
import express from 'express'
import { renderToString } from 'react-dom/server'
import Menu from './components/Menu'
```

```
import data from './assets/recipes.json'

global.React = React
const html = renderToString(<Menu recipes={data}/>)

const logger = (req, res, next) => {
  console.log(`${req.method} request for '${req.url}'`)
  next()
}

const sendHTMLPage = (req, res) =>
  res.status(200).send(`
<!DOCTYPE html>
<html>
  <head>
    <title>React Recipes App</title>
  </head>
  <body>
    <div id="react-container">${html}</div>
  </body>
</html>
`)

const app = express()
  .use(logger)
  .use(sendHTMLPage)

app.listen(3000, () =>
  console.log(`Recipe app running at 'http://localhost:3000'`)
)
```

Сначала мы импортируем `react`, метод `renderToString`, компонент `Menu` и несколько рецептов для наших исходных данных. Библиотека `React` представлена глобально, поэтому метод `renderToString` может работать правильно.

Затем с помощью вызова метода `renderToString` получается код HTML, отправляемый компоненту `Menu`.

И наконец, мы можем создать новую функцию связующего кода `sendHTMLPage`, которая откликается на все запросы строкой HTML-кода. В эту строку помещается отображенный на сервере код HTML в заготовке, необходимой для создания страницы.

Теперь при запуске этого приложения и переходе в браузере на `http://localhost:3000` вы увидите отображенные рецепты. В данный ответ нам не требуется включать какой-либо код JavaScript. Рецепты уже находятся на странице в виде кода HTML.

На данный момент у нас есть компонент `Menu`, отображаемый сервером. Наше приложение пока не приобрело изоморфности, поскольку компонент отображается только на сервере. Для придания изоморфности добавим к ответу небольшой фрагмент кода JavaScript, чтобы тот же самый компонент мог быть отображен в браузере.

Создадим файл `index-client.js`, который будет запускаться в браузере:

```
import React from 'react'
import { render } from 'react-dom'
import Menu from './components/Menu'

window.React = React
alert('bundle loaded, Rendering in browser')

render(
  <Menu recipes={__DATA__} />,
  document.getElementById("react-container")
)

alert('render complete')
```

Файл будет отображать тот же компонент `Menu` с теми же данными рецептов. Мы знаем, что данные те же самые, поскольку они уже будут включены в наш ответ в виде строки. Когда браузер загружает этот сценарий, переменная `__DATA__` уже будет существовать в глобальной области видимости. Чтобы увидеть моменты отображения браузером пользовательского интерфейса, задействуют методы `alert`.

Нам нужно собрать этой файл, `client.js`, в пакет, который может применяться браузером. Здесь сборка будет обрабатываться на основе базовой конфигурации `Webpack`.

Не забудьте установить пакет `webpack`; пакет `babel` и все необходимые предварительные настройки уже установлены:

```
npm install webpack --save-dev
```

Здесь сборка будет обрабатываться базовой конфигурацией `Webpack`:

```
var webpack = require("webpack")

module.exports = {
  entry: "./index-client.js",
  output: {
    path: "assets",
    filename: "bundle.js"
  },
  module: {
    rules: [
      {
        test: /\.js$/,
        exclude: /(node_modules)/,
        loader: 'babel-loader',
        query: {
          presets: ['env', 'stage-0', 'react']
        }
      }
    ]
  }
}
```

Нам нужно создавать клиентскую сборку при каждом запуске приложения, поэтому к файлу `package.json` следует добавить сценарий, предвещающий запуск:

```
"scripts": {
  "prestart": "./node_modules/.bin/webpack --progress",
  "start": "./node_modules/.bin/babel-node index-server.js"
},
```

Последним шагом будет внесение изменений в сервер. Исходные данные `__DATA__` нужно вписать в ответ в виде строки и, кроме того, включить тег `script` со ссылкой на наш клиентский пакет. И напоследок заставить сервер отправить статические файлы из каталога `./assets/`:

```
const sendHTMLPage = (req, res) =>
  res.status(200).send(`
<!DOCTYPE html>
<html>
  <head>
    <title>React Recipes App</title>
  </head>
  <body>
    <div id="react-container">${html}</div>
    <script>
      window.__DATA__ = ${JSON.stringify(data)}
    </script>
    <script src="bundle.js"></script>
  </body>
</html>
`)
```

```
const app = express()
  .use(logger)
  .use(express.static('./assets'))
  .use(sendHTMLPage)
```

Теги `script` были добавлены непосредственно в ответ. Данные вписаны в первый тег `script`, а пакет загружен во втором таком теге. Кроме того, в наш конвейер ответа был добавлен связующий код. Когда запрашивается файл `/bundle.js`, связующий код `express.static` отправит ответ с этим файлом вместо отображенного на сервере кода HTML, поскольку данный файл находится в папке `./assets`.

Теперь компоненты React отображаются изоморфно, сначала на сервере, а затем в браузере. При запуске этого приложения перед и после отображения компонентов в браузере вы увидите диалоговые окна с уведомлениями. Обратите внимание: прежде чем вы закроете первое уведомление, уже появится содержимое. Дело в том, что оно изначально отображено на сервере.

Может показаться неразумным отображать одно и то же содержимое дважды, но здесь есть свои преимущества. Это приложение отображает одно и то же содержимое во всех браузерах, даже если JavaScript отключен. Поскольку содержимое загружено в результате выполнения начального запроса, ваш сайт запустится быстрее и доставит запрашиваемый контент вашим мобильным пользователям

в более сжатые сроки¹. Им не придется ждать, пока мобильный процессор отобразит UI, — он уже будет на своем месте. Кроме того, это приложение вбирает в себя все преимущества одностраничного приложения. Изоморфные приложения React дают вам все самое лучшее из двух миров.

Универсальный органайзер цветов

В первых пяти главах мы занимались разработкой приложения органайзера цветов. На данный момент для него у нас наработана весьма обширная база программного кода, и весь этот код запускается в браузере. Мы запрограммировали компоненты React, хранилище Redux и целый ворох создателей действий и вспомогательных функций. Мы даже вставили React Router. У нас уже есть большой объем кода, который можно повторно использовать при создании веб-сервера.

Создадим для данного приложения Express-сервер и попробуем повторно применить как можно больший объем имеющегося кода. Сначала нам понадобится модуль, настраивающий экземпляр Express-приложения, поэтому создадим файл `./server/app.js`:

```
import express from 'express'
import path from 'path'
import fs from 'fs'

const fileAssets = express.static(
  path.join(__dirname, '../..../dist/assets')
)

const logger = (req, res, next) => {
  console.log(`${req.method} request for '${req.url}'`)
  next()
}

const respond = (req, res) =>
  res.status(200).send(`
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Universal Color Organizer</title>
  </head>
  <body>
    <div id="react-container">ready...</div>
  </body>
</html>
`)
```

¹ Farmer A. H. Should I use React Server-Side Rendering? <http://andrewfarmer.com/server-side-render/>.

```
export default express()
  .use(logger)
  .use(fileAssets)
  .use(respond)
```

Этот модуль является отправной точкой для нашего универсального приложения. В конфигурации Express используется связующий код для регистрации и файловых ресурсов, и в конечном итоге он отвечает на каждый запрос отправкой HTML-страницы.



Поскольку обслуживание HTML происходит напрямую из этого файла, вам следует удалить файл `./dist/index.html`. Если этот файл не удалить, он будет обработан еще до получения ответа на запрос.

Webpack позволяет выполнять импорт таких ресурсов, как CSS или файлы изображений, но среда Node.js не будет знать, как обращаться с этим импортом. Нам придется использовать библиотеку `ignore-styles`, чтобы добиться игнорирования любых импортируемых SCSS-инструкций. Установим `ignore-styles`:

```
npm install ignore-styles -save
```

В файле `./src/server/index.js` мы задействуем экземпляр Express-приложения и запустим сервер. Этот файл представляет точку входа для нашего сервера Node.js:

```
import React from 'react'
import ignoreStyles from 'ignore-styles'
import app from './app'

global.React = React

app.set('port', process.env.PORT || 3000)
  .listen(
    app.get('port'),
    () => console.log('Color Organizer running')
  )
```

Этот файл добавляет React к глобальному экземпляру и запускает сервер. Кроме того, мы включили модуль `ignore-styles`, заставляющий игнорировать соответствующий импорт, чтобы компоненты можно было отображать в Node.js без последствий в виде ошибок.

Теперь у нас есть отправная точка: базовая конфигурация Express-приложения. Всякий раз, когда понадобится добавлять к серверу новые функции, они должны будут проложить свой путь в данный модуль конфигурации приложения.

Во всем остальном материале данной главы мы пройдемся по этому Express-приложению. Код будет использоваться универсальным образом, чтобы создать как изоморфную, так и универсальную версию органайзера цветов.

Универсальный Redux

Весь код JavaScript в библиотеке Redux является универсальным. Ваши преобразователи написаны на JavaScript и не должны содержать код, зависящий от какой-либо среды окружения. Библиотека Redux была спроектирована в целях использования в качестве контейнера состояния для браузерных приложений, но она может применяться во всех типах приложений Node.js, включая приложения интерфейса командной строки, серверы, а также обычные приложения.

У нас уже имеется код для хранилища Redux. Оно будет служить для записи изменений состояния в JSON-файл на сервере.

Сначала нужно внести изменения в `storeFactory`, предоставив коду возможность работать в изоморфном режиме. На данный момент `storeFactory` включает связующий код регистрации, который станет в Node.js вызывать ошибки, поскольку в нем используются методы `console.groupCollapsed` и `console.groupEnd`. Эти методы недоступны в Node.js. Если мы создаем хранилища на сервере, то нужно будет задействовать другой код регистрации:

```
import { colors } from './reducers'
import {
  createStore, combineReducers, applyMiddleware
} from 'redux'

const clientLogger = store => next => action => {
  let result
  console.groupCollapsed("dispatching", action.type)
  console.log('prev state', store.getState())
  console.log('action', action)
  result = next(action)
  console.log('next state', store.getState())
  console.groupEnd()
  return result
}

const serverLogger = store => next => action => {
  console.log('\n dispatching server action\n')
  console.log(action)
  console.log('\n')
  return next(action)
}

const middleware = server =>
  (server) ? serverLogger : clientLogger

const storeFactory = (server = false, initialState = {}) =>
  applyMiddleware(middleware)(createStore)(
    combineReducers({colors}),
    initialState
  )

export default storeFactory
```


Теперь код `storeFactory` является изоморфным. Мы создали связующий код `Redux` для регистрации действий на сервере. При вызове `storeFactory` мы сообщим этому методу о том, какой тип хранилища нам нужен, и к новому экземпляру хранилища будет добавлен соответствующий регистратор.

Теперь воспользуемся изоморфным `storeFactory` для создания экземпляра `serverStore`. В верхней части конфигурации `Express` нужно импортировать `storeFactory` и исходные данные состояния. Для создания хранилища с исходным состоянием из `JSON`-файла можно применить `storeFactory`:

```
import storeFactory from '../store'
import initialState from '../../data/initialState.json'

const serverStore = storeFactory(true, initialState)
```

Теперь у нас есть экземпляр хранилища, которое будет запускаться на сервере.

При каждой диспетчеризации действия к этому экземпляру нужно убедиться в обновлении файла `initialState.json`. Используя метод `subscribe`, можно отслеживать изменения состояния и сохранять новый `JSON`-файл при каждом изменении:

```
serverStore.subscribe(() =>
  fs.writeFile(
    path.join(__dirname, '../../data/initialState.json'),
    JSON.stringify(serverStore.getState()),
    error => (error) ?
      console.log("Error saving state!", error) :
      null
  )
)
```

После диспетчеризации действий происходит сохранение нового состояния в файле `initialState.json`, для чего используется модуль `fs`.

Теперь основным источником истины является `serverStore`. Для получения текущего и самого актуального списка цветов с ним придется связываться любым запросам. Мы еще добавим часть связующего кода, который будет добавлять серверное хранилище к конвейеру запросов, чтобы его можно было использовать в ходе запроса другому связующему коду:

```
const addStoreToRequestPipeline = (req, res, next) => {
  req.store = serverStore
  next()
}

export default express()
  .use(logger)
  .use(fileAssets)
  .use(addStoreToRequestPipeline)
  .use(htmlResponse)
```

Теперь любой метод связующего кода, следующий после `addStoreToRequestPipeline`, будет иметь доступ к хранилищу в объекте запроса `request`. Мы воспользовались Redux универсальным образом. Абсолютно одинаковый код хранилища, включая наши преобразователи, будет запускаться в нескольких средах.



С созданием веб-серверов для крупных приложений связан ряд сложностей, которые в данном примере не рассматриваются. Сохранение данных в JSON-файле является быстрым способом их длительного хранения, но приложения, предназначенные для практического использования, задействуют настоящие базы данных. Применение Redux — вполне возможный выход, соответствующий требованиям отдельных приложений. Но существуют сложности, связанные с процессами разветвляющихся узлов, требующие решений в более крупных приложениях. Стоит изучить такие решения, как Firebase и облачные провайдеры, предоставленные с целью помочь в работе с базами данных, которые могут плавно масштабироваться.

Универсальная маршрутизация

В предыдущей главе к органайзеру цветов был добавлен пакет-маршрутизатор `react-router-dom`. Он решает, какой компонент отображать, основываясь на текущем местоположении окна браузера. Маршрутизатор может также выполнять отображение на сервере, для чего ему нужно просто предоставить местоположение или маршрут.

До сих пор мы использовали компонент `HashRouter`. Маршрутизатор автоматически добавляет перед каждым маршрутом знак `#`. Чтобы применить маршрутизатор в изоморфном режиме, нужно заменить `HashRouter` компонентом `BrowserRouter`, который удаляет предшествующий знак `#` из маршрутов.

При отображении приложения следует заменить `HashRouter` компонентом `BrowserRouter`:

```
import { BrowserRouter } from 'react-router-dom'

...

render(
  <Provider store={store}>
    <BrowserRouter>
      <App />
    </BrowserRouter>
  </Provider>,
  document.getElementById('react-container')
)
```

Теперь органайзер цветов больше не ставит перед каждым маршрутом знак решетки. На данный момент он сохраняет свою работоспособность. Запустите органай-

зер и выберите один цвет. Будет отображен контейнер `Color`, а фон всего экрана изменится с помощью компонента `ColorDetails`.

Адресная строка должна приобрести следующий вид:

```
http://localhost:3000/8658c1d0-9eda-4a90-95e1-8001e8eb6036
```

Теперь перед маршрутом нет знака `#`. Обновим страницу в браузере:

```
Cannot GET /8658c1d0-9eda-4a90-95e1-8001e8eb6036
```

Это обновление заставляет браузер отправить на сервер GET-запрос, используя текущий маршрут. Знак `#` использовался в целях помешать отправить данный запрос. Мы задействуем `BrowserRouter`, поскольку нам нужно, чтобы GET-запрос был отправлен на сервер. Для отображения маршрутизатора на сервер требуется местоположение, то есть необходим маршрут. Он будет использоваться на сервере, чтобы сообщить маршрутизатору о необходимости отобразить контейнер `Color`. Когда требуется изоморфное отображение маршрутов, применяется компонент `BrowserRouter`.

Теперь, когда известно, какое содержимое запрашивает пользователь, задействуем эти знания для отображения UI на сервере. Чтобы отобразить маршрутизатор на сервер, придется внести ряд существенных изменений в Express-конфигурацию. Для начала следует импортировать несколько модулей:

```
import { Provider } from 'react-redux'
import { compose } from 'redux'
import { renderToString } from 'react-dom/server'
import { StaticRouter } from 'react-router-dom'
```

Понадобятся компонент `Provider`, функция `compose`, функция `renderToString` и компонент `StaticRouter`. На сервере компонент `StaticRouter` используется при необходимости отобразить дерево компонентов на строку.

Чтобы создать HTML-ответ, требуется выполнить три шага.

1. Создать хранилище, запускаемое на стороне клиента с помощью данных из `serverStore`.
2. Отобразить дерево компонентов в виде кода HTML, используя `StaticRouter`.
3. Создать HTML-страницу, которая будет отправлена клиенту.

Для каждого из шагов мы создадим по одной функции и сведем их в одну по имени `htmlResponse`:

```
const htmlResponse = compose(
  buildHTMLPage, // Шаг 3
  renderComponentsToHTML, // Шаг 2
  makeClientStoreFrom(serverStore) // Шаг 1
)
```

В этой композиции `makeClientStoreFrom(serverStore)` является функцией высшего порядка. Изначально вызывается один раз с `serverStore`. Она возвращает

функцию, которая будет вызвана при каждом запросе. Возвращенная функция будет всегда иметь доступ к `serverStore`.

Когда вызывается метод `htmlResponse`, он ожидает передачи одного аргумента: `url`, то есть URL, запрошенного пользователем. Выполняя шаг 1, мы создадим функцию высшего порядка, которая упакует `url` вместе с новым хранилищем на стороне клиента `store`, созданным с помощью текущего состояния `serverStore`. И `store`, и `url` передаются следующей функции, что является выполнением шага 2, в едином объекте:

```
const makeClientStoreFrom = store => url =>
  ({
    store: storeFactory(false, store.getState()),
    url
  })
```

Выходные данные из функции `makeClientStoreFrom` становятся входными данными для функции `renderComponentToHTML`. Эта функция ожидает `url` и `store`, упакованные в один аргумент:

```
const renderComponentsToHTML = ({url, store}) =>
  ({
    state: store.getState(),
    html: renderToString(
      <Provider store={store}>
        <StaticRouter location={url} context={{}}>
          <App />
        </StaticRouter>
      </Provider>
    )
  })
```

Функция `renderComponentsToHTML` возвращает объект с двумя свойствами: `state` и `html`. Свойство `state` получается из нового клиентского хранилища, а свойство `html` создается методом `renderToString`. Поскольку приложение по-прежнему используется в браузере Redux, компонент `Provider` отображается в виде корневого компонента, а новое клиентское хранилище передается ему как свойство.

Компонент `StaticRouter` служит для отображения пользовательского интерфейса на основе запрошенного местоположения. Компоненту `StaticRouter` требуется `location` и `context`. Свойству `location` передается запрошенный `url`, а свойству `context` — пустой объект. Когда эти компоненты отображаются в HTML-строку, `location` будет приниматься в расчет и `StaticRouter` отобразит правильные маршруты.

Эта функция возвращает два обязательных для создания страницы компонента: текущее состояние органайзера и пользовательский интерфейс, отображенный в HTML-строку.

И `state`, и `html` могут применяться в последней составной функции `buildHTMLPage`:

```

const buildHTMLPage = ({html, state}) => `
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Universal Color Organizer</title>
  </head>
  <body>
    <div id="react-container">${html}</div>
    <script>
      window.__INITIAL_STATE__ = ${JSON.stringify(state)}
    </script>
    <script src="/bundle.js"></script>
  </body>
</html>
`

```

Теперь наше представление цветов имеет изоморфный характер. Отображение пользовательского интерфейса выполняется на сервере, а затем отправляется клиенту в виде текста. В данный ответ также будет непосредственно вставлено исходное состояние хранилища.

Изначально браузер показывает UI, полученный в HTML-ответе. В момент загрузки пакета интерфейс отображается заново, и с этого момента клиентская сторона берет все на себя. Теперь взаимодействие с пользователем, включая переходы, будет происходить на стороне клиента. Одностраничное приложение станет работать в обычном для себя режиме вплоть до обновления содержимого страницы браузера, и с того момента процесс отображения на сервере начнется снова.

Весь файл целиком со всем текущим кодом из модуля Express-приложения выглядит следующим образом:

```

import express from 'express'
import path from 'path'
import fs from 'fs'
import { Provider } from 'react-redux'
import { compose } from 'redux'
import { StaticRouter } from 'react-router-dom'
import { renderToString } from 'react-dom/server'
import App from '../components/App'
import storeFactory from '../store'
import initialState from '../../data/initialState.json'

const fileAssets = express.static(
  path.join(__dirname, '../../dist/assets')
)

const serverStore = storeFactory(true, initialState)

serverStore.subscribe(() =>
  fs.writeFile(
    path.join(__dirname, '../../data/initialState.json'),
    JSON.stringify(serverStore.getState()),
    error => (error) ?

```

```
        console.log("Error saving state!", error) :
        null
    )
)

const logger = (req, res, next) => {
  console.log(`${req.method} request for '${req.url}'`)
  next()
}

const addStoreToRequestPipeline = (req, res, next) => {
  req.store = serverStore
  next()
}

const makeClientStoreFrom = store => url =>
  ({
    store: storeFactory(false, store.getState()),
    url
  })

const renderComponentsToHTML = ({url, store}) =>
  ({
    state: store.getState(),
    css: defaultStyles,
    html: renderToString(
      <Provider store={store}>
        <StaticRouter location={url} context={{}}>
          <App />
        </StaticRouter>
      </Provider>
    )
  })

const buildHTMLPage = ({html, state}) => `
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Universal Color Organizer</title>
  </head>
  <body>
    <div id="react-container">${html}</div>
    <script>
      window.__INITIAL_STATE__ = ${JSON.stringify(state)}
    </script>
    <script src="/bundle.js"></script>
  </body>
</html>
`

const htmlResponse = compose(
  buildHTMLPage,
  renderComponentsToHTML,
  makeClientStoreFrom(serverStore)
)
```

```

const respond = (req, res) =>
  res.status(200).send(htmlResponse(req.url))

export default express()
  .use(logger)
  .use(fileAssets)
  .use(addStoreToRequestPipeline)
  .use(respond)
    
```

Теперь приложение позволяет пользователям делать закладки URL и отправлять адреса другим пользователям при изоморфном режиме отображения. Маршрутизатор на основе URL решает, какое содержимое отобразить. Он делает это на сервере, следовательно, посетители сайта могут получить быстрый доступ к содержимому.

Изоморфные приложения взяли все самое лучшее из двух миров: они могут воспользоваться преимуществами скорости, управления и безопасности, которые обеспечивает процесс отображения на сервере, а также выгодами, вытекающими из низкой пропускной способности сети и скорости передачи данных, свойственной одностраничным приложениям. Итак, в конечном счете изоморфное React-приложение свелось к одностраничному с отображением на сервере; так закладывается основа для создания эффективных приложений, нетребовательных к сетевым ресурсам и в то же время обладающих быстротой и продуктивностью.

Включение стилей, отображаемых на сервере. Сейчас отображение кода HTML происходит на сервере, но таблицы CSS не отображаются до тех пор, пока пакет не будет загружен в браузер. В результате на экране наблюдается странное мерцание. Вначале, до загрузки CSS, на экране браузера появится все содержимое без стилового оформления. Когда в браузере отключен JavaScript, пользователи вообще не увидят задаваемые таблицами CSS стили, поскольку они встроены в пакет JavaScript.

Решение состоит в добавлении стилей непосредственно в ответ. Для этого сначала нужно будет извлечь код CSS из Webpack-пакета в отдельный файл, что потребует установки пакета `extract-text-webpack-plugin`:

```
npm install extract-text-webpack-plugin
```

Кроме того, понадобится затребовать этот пакет в вашей конфигурации Webpack:

```

var webpack = require("webpack")
var ExtractTextPlugin = require("extract-text-webpack-plugin")
var OptimizeCss = require('optimize-css-assets-webpack-plugin')
    
```

Затем в Webpack-конфигурации нужно заменить загрузчики CSS и SCSS теми, которые используют `ExtractTextPlugin`:

```

{
  test: /\.css$/,
    
```

```

    loader: ExtractTextPlugin.extract({
      fallback: "style-loader",
      use: [
        "style-loader",
        "css-loader",
        {
          loader: "postcss-loader",
          options: {
            plugins: () => [require("autoprefixer")]
          }
        }
      ]
    })
  },
  {
    test: /\.scss/,
    loader: ExtractTextPlugin.extract({
      fallback: "style-loader",
      use: [
        "css-loader",
        {
          loader: "postcss-loader",
          options: {
            plugins: () => [require("autoprefixer")]
          }
        },
        "sass-loader"
      ]
    })
  }
}

```

Вдобавок нужно будет включить этот дополнительный модуль в конфигурацию, а именно в массив дополнительных модулей. Здесь, когда дополнительный модуль включен, указывается имя файла CSS, из которого выполняется извлечение данных:

```

plugins: [
  new ExtractTextPlugin("bundle.css"),
  new OptimizeCss({
    assetNameRegExp: /\.optimize\.css$/g,
    cssProcessor: require('cssnano'),
    cssProcessorOptions: {
      discardComments: {removeAll: true}
    },
    canPrint: true
  })
]

```

Теперь, после запуска Webpack, кода CSS не окажется в пакете JavaScript, вместо этого все инструкции CSS будут извлечены из отдельного файла `./assets/bundle.css`.

Нужно внести изменения и в конфигурацию Express. При запуске органайзера код CSS сохраняется в виде строки с глобальной доступностью. Для считывания

содержимого текстового файла в переменную `staticCSS` можно воспользоваться файловой системой или модулем `fs`:

```
const staticCSS = fs.readFileSync(
  path.join(__dirname, '.././dist/assets/bundle.css')
)
```

Теперь следует внести изменения в функцию `buildHTMLPage`, чтобы записать CSS непосредственно в ответ внутри тега `<style>`:

```
const buildHTMLPage = ({html, state}) => `
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Universal Color Organizer</title>
    <style>${staticCSS}</style>
  </head>
  <body>
    <div id="react-container">${html}</div>
    <script>
      window.__INITIAL_STATE__ = ${JSON.stringify(state)}
    </script>
    <script src="/bundle.js"></script>
  </body>
</html>
`
```

Теперь код CSS непосредственно встроен в ответ, в результате чего странные мерцания содержимого, лишнего стилового оформления, прекратятся. Когда JavaScript отключен, стили останутся на месте.

Теперь у нас есть изоморфный органайзер цветов, в котором в обеих средах совместно используется большой объем универсального кода JavaScript. Изначально органайзер отображается на сервере, но после завершения загрузки страницы отображается и в браузере. Когда основная роль переходит к браузеру, органайзер ведет себя как одностраничное приложение.

Обмен данными с сервером

На данный момент органайзер цветов отображает UI на сервере и еще раз в браузере. Как только браузер берет все на себя, органайзер работает как одностраничное приложение. Действия пользователя проходят диспетчеризацию локально, локальное состояние изменяется, и интерфейс изменяется тоже локально. Все вполне успешно работает в браузере, но прошедшие диспетчеризацию действия не возвращаются на сервер.

В следующем разделе мы не только обеспечим сохранение этих данных на сервере, но и сделаем так, чтобы сами объекты действий создавались на сервере и проходили диспетчеризацию в оба хранилища.

Выполнение действий на сервере

Для работы с данными органайзера цветов мы подключим REST API (технологии репрезентативной передачи состояния API). Действия будут инициироваться на стороне клиента, выполняться на сервере, а затем проходить диспетчеризацию в оба хранилища. Хранилище на стороне сервера, `serverStore`, будет сохранять новое состояние в JSON-формате, а хранилище на стороне клиента будет запускать обновление пользовательского интерфейса. Оба хранилища будут выполнять диспетчеризацию одних и тех же действий в универсальном режиме (рис. 12.2).

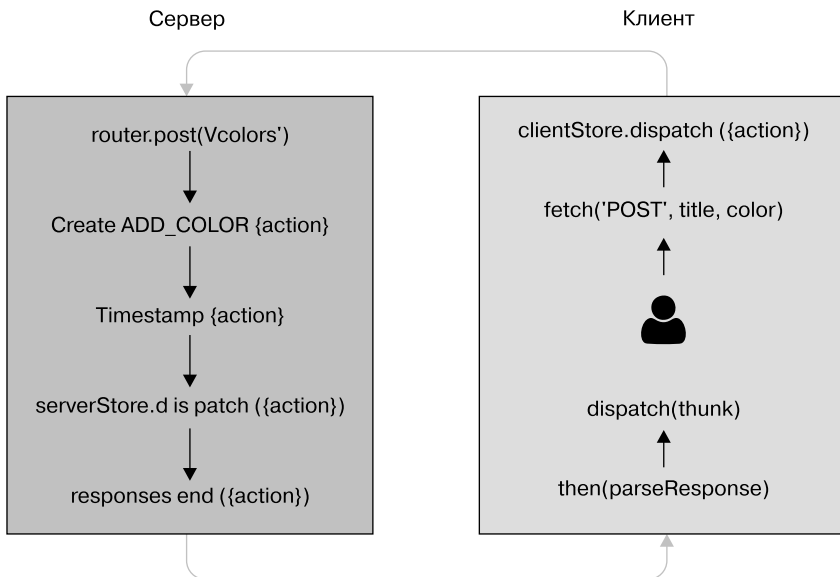


Рис. 12.2. Создание универсального действия

Изучим на примере весь процесс диспетчеризации действия `ADD_COLOR` в предлагаемом решении.

1. Диспетчеризация создателя действия `addColor()` с новым названием и цветом.
2. Отправка данных на сервер в новом `POST`-запросе.
3. Создание и диспетчеризация нового действия по добавлению цвета `ADD_COLOR` на сервере.
4. Отправка действия `ADD_COLOR` в теле ответа.
5. Парсинг тела ответа и диспетчеризация действия `ADD_COLOR` на стороне клиента.

Первым делом нужно произвести сборку REST API. Создадим новый файл `./src/server/color-api.js`.

Каждое созданное действие обрабатывается одинаково: выполняется его диспетчеризация на сервер, а затем оно отправляется клиенту. Создадим функцию, выполняющую диспетчеризацию действия на серверное хранилище `Store` и отправляющую действие клиенту, используя объект ответа:

```
const dispatchAndRespond = (req, res, action) => {
  req.store.dispatch(action)
  res.status(200).json(action)
}
```

При наличии действия эту функцию можно применять для диспетчеризации этого действия и отправки ответа на сторону клиента.

Нам нужно создать ряд конечных точек HTTP, используя маршрутизатор `Express Router`, способный обрабатывать различные HTTP-запросы. Мы создадим маршруты для обработки запросов `GET`, `POST`, `PUT` и `DELETE` на маршруте `/api/colors`. Для создания этих маршрутов можно задействовать `Express Router`. Каждый маршрут будет содержать логику для создания отдельного объекта действия и отправки его функции `dispatchAndRespond` вместе с объектами запроса и ответа:

```
import { Router } from 'express'
import { v4 } from 'uuid'

const dispatchAndRespond = (req, res, action) => {
  req.store.dispatch(action)
  res.status(200).json(action)
}

const router = Router()

router.get("/colors", (req, res) =>
  res.status(200).json(req.store.getState().colors)
)

router.post("/colors", (req, res) =>
  dispatchAndRespond(req, res, {
    type: "ADD_COLOR",
    id: v4(),
    title: req.body.title,
    color: req.body.color,
    timestamp: new Date().toString()
  })
)

router.put("/color/:id", (req, res) =>
  dispatchAndRespond(req, res, {
    type: "RATE_COLOR",
    id: req.params.id,
    rating: parseInt(req.body.rating)
  })
)

router.delete("/color/:id", (req, res) =>
  dispatchAndRespond(req, res, {
```

```

      type: "REMOVE_COLOR",
      id: req.params.id
    })
  )
)

```

```
export default router
```

Каждая функция, добавленная к объекту маршрутизатора, обрабатывает отдельный запрос для `http://localhost:3000/api/{route}`.

- ❑ GET `'/colors'` — отвечает выдачей текущего массива цветов из состояния на стороне сервера. Этот маршрут добавляется только для того, чтобы у нас была возможность увидеть список цветов, на стороне клиента он не используется.
- ❑ POST `'/colors'` — создает новый объект действия с цветами и отправляет его функции `dispatchAndRespond`.
- ❑ PUT `'/color/:id'` — изменяет рейтинг цвета. Идентификатор цвета берется из параметров маршрута и применяется в новом объекте действия.
- ❑ DELETE `'/color/:id'` — удаляет цвет на основе идентификатора, переданного в параметрах маршрутизации.

Теперь, при наличии определенных маршрутов, их нужно добавить к конфигурации Express-приложения. Сначала установим Express-модуль `body-parser`:

```
npm install body-parser --save
```

Модуль используется для парсинга тел поступающих запросов и получения всех переменных, отправленных маршрутам. Из данных на стороне клиента нужно получить новую информацию о цвете и его оценке. Следует добавить этот связующий код к конфигурации Express-приложения. Импортируем `body-parser` и новые маршруты в файл `./server/app.js`:

```
import bodyParser from 'body-parser'
import api from './color-api'
```

Добавим связующий код `bodyParser` и API к Express-приложению. Важно добавить `bodyParser` до API, чтобы данные могли быть разобраны парсером к тому времени, как API обработает запрос:

```
export default express()
  .use(logger)
  .use(fileAssets)
  .use(bodyParser.json())
  .use(addStoreToRequestPipeline)
  .use('/api', api)
  .use(matchRoutes)
```

Теперь `bodyParser.json()` выполняет парсинг тела поступающего запроса, имеющего формат JSON. Модуль `color-api` добавлен к конвейеру и настроен на ответ по любому маршруту, имеющему префикс `/api`. Например, следующий URL может быть использован для получения текущего массива цветов в JSON-формате: `http://localhost:3000/api/colors`.

Теперь, когда у Express-приложения имеются конечные точки, способные реагировать на HTTP-запросы, мы готовы изменить создатели действий на стороне клиента, чтобы получить возможность обмениваться данными с этими конечными точками.

Действия с Redux Thunks

Одной из проблем обмена данными клиента с сервером является запаздывание или задержка данных, ощущаемая в ходе ожидания ответа после отправки запроса. Прежде чем получить возможность диспетчеризации действия, создателям действий приходится ждать ответа, поскольку в решении само действие отправляется клиенту с сервера. Для Redux имеется связующий код, который может оказать помощь в асинхронных действиях: он называется `redux-thunk`.

В следующем разделе мы переделаем создатели действий под использование `redux-thunk`. Эти создатели, которые называются переходниками (`thunk`-функциями), позволят дождаться ответа сервера, прежде чем выполнять локальную диспетчеризацию действия. `Thunk`-функции являются функциями высшего порядка. Вместо объектов действий они возвращают другие функции. Установим `redux-thunk`:

```
npm install redux-thunk --save
```

Модуль `redux-thunk` относится к связующему коду, который следует встроить в `storeFactory`. Сначала в верхней части файла `./src/store/index.js` нужно разместить инструкцию импортирования `redux-thunk`:

```
import thunk from 'redux-thunk'
```

В `storeFactory` имеется функция под названием `middleware`. Она возвращает связующий код, который должен быть встроен в новое хранилище в единый массив. К этому массиву можно добавить любой связующий код Redux. Каждый элемент будет развернут в аргументы функции `applyMiddleware`:

```
const middleware = server => [
  (server) ? serverLogger : clientLogger,
  thunk
]

const storeFactory = (server = false, initialState = {}) =>
  applyMiddleware(...middleware(server))(createStore)(
    combineReducers({colors}),
    initialState
  )

export default storeFactory
```

Посмотрим на текущий создатель действий для добавления цветов:

```
export const addColor = (title, color) =>
  ({
```

```

    type: "ADD_COLOR",
    id: v4(),
    title,
    color,
    timestamp: new Date().toString()
  })
}

```

...

```
store.dispatch(addColor("jet", "#000000"))
```

Этот создатель возвращает объект, представляющий собой действие `addColor`. Данный объект тут же проходит диспетчеризацию в хранилище. А теперь посмотрим на `thunk`-версию `addColor`:

```

export const addColor = (title, color) =>
  (dispatch, getState) => {
    setTimeout(() =>
      dispatch({
        type: "ADD_COLOR",
        index: getState().colors.length + 1,
        timestamp: new Date().toString()
        title,
        color
      }
    ),
    2000
  )
}

```

...

```
store.dispatch(addColor("jet", "#000000"))
```

Несмотря на то что оба создателя действий проходят диспетчеризацию абсолютно одинаково, `thunk`-версия возвращает не действие, а функцию. Это функция обратного вызова, получающая в виде аргументов методы `dispatch` и `getState`, принадлежащие хранилищу. По готовности можно выполнить диспетчеризацию действия. В данном примере метод `setTimeout` используется для создания двухсекундной задержки перед диспетчеризацией действия по добавлению нового цвета.

Кроме `dispatch`, у `thunk`-функций также имеется доступ к принадлежащему хранилищу методу `getState`. В данном примере он используется для создания поля `index` на основе текущего количества цветов, имеющих в состоянии. Эта функция может пригодиться, когда наступит время создавать действия, зависящие от данных, получаемых из хранилища.



Представлять в виде `thunk`-версий все ваши создатели действий не обязательно. Связующий код `redux-thunk` понимает разницу между объектами действий и `thunk`-объектами. Диспетчеризация объектов действий выполняется без задержки.

У `thunk`-функций имеется еще одно преимущество. Они могут вызывать метод `dispatch` или метод `getState` в асинхронном режиме произвольное количество раз, и на них не накладываются ограничения по диспетчеризации действий только одного типа. В следующем примере `thunk`-функция сразу же выполняет диспетчеризацию действия `RANDOM_RATING_STARTED` и несколько раз повторно совершает диспетчеризацию действия `RATE_COLOR`, присваивающего конкретному цвету произвольный рейтинг:

```
export const rateColor = id =>
  (dispatch, getState) => {

    dispatch({ type: "RANDOM_RATING_STARTED" })
    setInterval(() =>
      dispatch({
        type: "RATE_COLOR",
        id,
        rating: Math.floor(Math.random()*5)
      }),
      1000
    )
  }

...

store.dispatch(
  rateColor("f9005b4e-975e-433d-a646-79df172e1dbb")
)
```

Эти `thunk`-функции приведены в качестве примеров. Создадим настоящую `thunk`-функцию, которую органайзер цветов будет использовать для замены текущих создателей действий.

Сначала создадим функцию `fetchThenDispatch`. В ней для отправки запроса к веб-сервису и автоматической диспетчеризации ответа применяется функция `isomorphic-fetch`:

```
import fetch from 'isomorphic-fetch'

const parseResponse = response => response.json()
const logError = error => console.error(error)
const fetchThenDispatch = (dispatch, url, method, body) =>
  fetch(
    url,
    {
      method,
      body,
      headers: { 'Content-Type': 'application/json' }
    }
  ).then(parseResponse)
  .then(dispatch)
  .catch(logError)
```

Функция `fetchThenDispatch` требует в качестве аргументов функцию `dispatch`, URL, метод HTTP-запроса и тело данного запроса. Затем эта информация используется

в функции `fetch`. Сразу после получения ответ будет разобран парсером, а затем пройдет диспетчеризацию. Все возможные ошибки отобразятся в консоли.

Мы задействуем функцию `fetchThenDispatch` в качестве вспомогательной при создании thunk-функций. Каждая такая функция будет отправлять к нашему API запрос, сопровождая его необходимыми данными. Поскольку наш API дает ответ в виде объектов действий, этот ответ тут же может пройти обработку парсером и диспетчеризацию:

```
export const addColor = (title, color) => dispatch =>
  fetchThenDispatch(
    dispatch,
    '/api/colors',
    'POST',
    JSON.stringify({title, color})
  )

export const removeColor = id => dispatch =>
  fetchThenDispatch(
    dispatch,
    `/api/color/${id}`,
    'DELETE'
  )

export const rateColor = (id, rating) => dispatch =>
  fetchThenDispatch(
    dispatch,
    `/api/color/${id}`,
    'PUT',
    JSON.stringify({rating})
  )
```

Thunk-версия `addColor` отправляет POST-запрос на `http://localhost:3000/api/colors` вместе с названием и шестнадцатеричным значением нового цвета. В ответ возвращается объект действия `ADD_COLOR`, который проходит обработку парсером и диспетчеризацию.

Thunk-версия `removeColor` отправляет в адрес API DELETE-запрос, сопровождая его в URL идентификатором удаляемого цвета. В ответ возвращается объект действия `REMOVE_COLOR`, который проходит обработку парсером и диспетчеризацию.

Thunk-версия `rateColor` отправляет в адрес API PUT-запрос. В URL в качестве параметра маршрута включается идентификатор цвета, получающего оценку, а новая оценка предоставляется в теле запроса. В ответ с сервера возвращается объект действия `RATE_COLOR`, обрабатываемый парсером как код формата JSON и проходящий диспетчеризацию в локальное хранилище.

Теперь, после запуска приложения, в консольном журнале регистрации можно будет увидеть действия, прошедшие диспетчеризацию в оба хранилища. Консоль браузера является частью инструмента разработчика, а консоль сервера представлена терминалом, на котором запущен сервер (рис. 12.3).

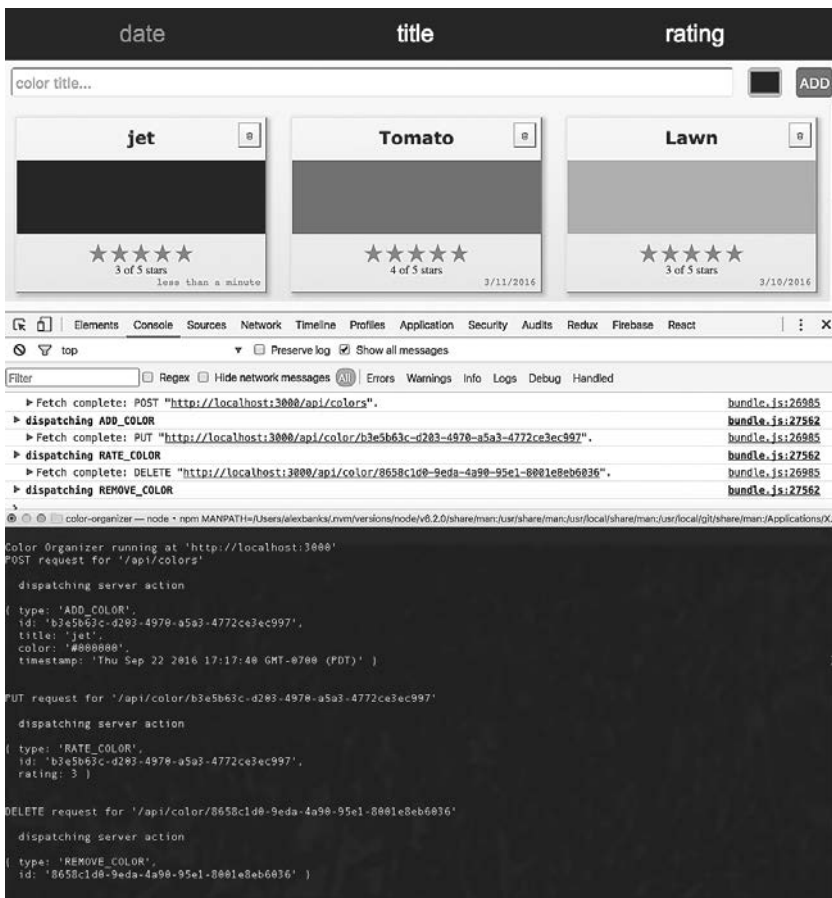


Рис. 12.3. Консоль браузера и консоль сервера

Использование Think-функций с веб-сокетами

Организатор цветов применяет технологию REST, чтобы обмениваться данными с сервером. Для отправки и получения данных think-функции можно использовать с веб-сокетами (websockets). Последние предоставляют возможность двустороннего подключения между клиентом и сервером. Веб-сокеты могут отправлять данные на сервер, а также позволяют серверу отправлять данные клиенту.

Один из способов работы с веб-сокетами и think-функциями — диспетчеризация создателя действия подключения `connect`. Допустим, нам нужно подключиться к серверу сообщений:

```
store.dispatch(connectToMessageSocket())
```

Thunk-функции могут вызывать метод `dispatch` любое количество раз. Можно создать thunk-функции, отслеживающие поступающие сообщения и выполняющие диспетчеризацию действий `NEW_MESSAGE` при их получении. В следующем фрагменте кода для подключения к серверу `socket.io` и отслеживания поступающих сообщений используется `socket.io-client`:

```
import io from 'socket.io-client'

const connectToChatSocket = () => dispatch => {
  dispatch({type: "CONNECTING"})

  let socket = io('/message-socket')

  socket.on('connect', () =>
    dispatch({type: "CONNECTED", id: socket.id})
  )

  socket.on('message', (message, user) =>
    dispatch({type: "NEW_MESSAGE", message, user})
  )
}

export default connectToMessageSocket
```

Сразу после вызова метода `connectToChatSocket` выполняется диспетчеризация действия `CONNECTING`. Затем предпринимается попытка подключиться к сокету сообщений. После успешного подключения сокет отреагирует на событие `connect`. Как только это произойдет, можно выполнить диспетчеризацию действия `CONNECTED` с информацией о текущем сокете.

Когда сервер отправляет новые сообщения, в сокете выдаются события сообщения. Действия `NEW_MESSAGE` могут проходить локальную диспетчеризацию при каждой отправке данному клиенту с сервера.

Thunk-функции могут работать с асинхронными процессами любого типа, включая веб-сокеты `socket.io`, `Firebase`, `setTimeouts`, переходы и анимации.

Практически каждому React-приложению, которое вы создадите, потребуется веб-сервер какого-либо типа. Иногда он будет необходим только для размещения вашего приложения. В иных ситуациях потребуется обмен данными с веб-сервисами. И это будут приложения с высоким уровнем трафика, которым нужно работать на многих платформах, для чего потребуются совершенно разные решения.

Усовершенствованный метод получения данных

При работе над созданием приложений с высоким уровнем трафика, обменивающихся данными с несколькими платформами, можно обратить внимание на такие

фреймворки, как Relay и GraphQL или Falcor. Они предоставляют более эффективные и рациональные решения для снабжения приложений только теми данными, которые ими запрашиваются. GraphQL (<http://graphql.org/>) — декларативное решение по запросу данных, разработанное в компании Facebook; может применяться для запроса данных из нескольких источников. GraphQL подходит любым типам языков и платформ. Relay (<https://facebook.github.io/relay/>) является библиотекой, также разработанной компанией Facebook, и получает данные для клиентских приложений путем привязки запросов GraphQL к компонентам React или React Native. Осваивать GraphQL и Relay сравнительно легко, но дело пойдет быстрее, если вам действительно нравится декларативное программирование.

Falcor (<https://netflix.github.io/falcor/>) — фреймворк, разработанный компанией Netflix, который также направлен на решение вопросов, связанных с получением и эффективным использованием данных. Как и GraphQL, Falcor позволяет запрашивать в одном месте данные от нескольких сервисов. Но в Falcor для запроса данных применяется JavaScript, что, вероятно, означает более легкое изучение этой утилиты разработчиками, пишущими на этом языке.

Ключевым подходом к разработке React-приложений является умение разбираться в том, когда и какие инструменты следует использовать. В вашем арсенале уже имеется немало инструментов, необходимых для создания надежных приложений. Но применяйте только те из них, что реально нужны. Если ваше приложение не зависит от большого объема данных, не пользуйтесь Redux. Для него вполне подойдет состояние React, которое отлично впишется в приложение разумного размера. Вашему приложению может вообще не понадобиться отображение на стороне сервера. И не нужно всерьез задумываться о его встраивании до тех пор, пока дело не дойдет до создания приложения с высоким уровнем интерактивности, использующим большой объем мобильного трафика.

Мы надеемся, что эта книга послужит вам хорошим справочником и полезным подспорьем при создании ваших собственных React-приложений. Конечно же, и сама библиотека React, и связанные с ней библиотеки еще будут изменяться, но при этом сохранят статус весьма надежных инструментов, придающих уверенность в том, что ими можно воспользоваться прямо сейчас. Создание приложения с помощью React, Redux и функционального, декларативного языка JavaScript представляется весьма увлекательным занятием, и нам уже не терпится увидеть созданное вами.

Алекс Бэнкс, Ева Порселло
React и Redux: функциональная веб-разработка

Перевел с английского Н. Вильчинский

Заведующая редакцией	<i>Ю. Сергиенко</i>
Руководитель проекта	<i>О. Сивченко</i>
Ведущий редактор	<i>Н. Гринчик</i>
Литературный редактор	<i>Н. Хлебина</i>
Художественный редактор	<i>С. Заматевская</i>
Корректоры	<i>Е. Павлович, Е. Рафалюк-Бузовская</i>
Верстка	<i>О. Богданович</i>

Изготовлено в России. Изготовитель: ООО «Прогресс книга».

Место нахождения и фактический адрес: 191123, Россия, город Санкт-Петербург,
улица Радищева, дом 39, корпус Д, офис 415. Тел.: +78127037373.

Дата изготовления: 11.2017. Наименование: книжная продукция. Срок годности: не ограничен.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12 —
Книги печатные профессиональные, технические и научные.

Подписано в печать 08.11.17. Формат 70×100/16. Бумага офсетная. Усл. п. л. 27,090. Тираж 1000. Заказ 0000.

Отпечатано в ОАО «Первая Образцовая типография». Филиал «Чеховский Печатный Двор».

142300, Московская область, г. Чехов, ул. Полиграфистов, 1.

Сайт: www.chpk.ru. E-mail: marketing@chpk.ru

Факс: 8(496) 726-54-10, телефон: (495) 988-63-87