ВЕБ-ПРИЛОЖЕНИЯ НА REACT, JSX, REDUX И GRAPHQL

React

Быстро

АЗАТ МАРДАН

Предисловие
Джона Сонмеза

MANNING

React Quickly

PAINLESS WEB APPS WITH REACT, JSX, REDUX, AND GRAPHQL

AZAT MARDAN
FOREWORD BY JOHN SONMEZ

MANNING
SHELTER ISLAND

АЗАТ МАРДАН

Предисловие
Джона Сонмеза

React
Быстро

ВЕБ-ПРИЛОЖЕНИЯ НА REACT, JSX, REDUX И GRAPHQL

ПИТЕР®

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

2019

ББК 32.988.02-018
УДК 004.738.5
М25

Мардан Азат

М25 React быстро. Веб-приложения на React, JSX, Redux и GraphQL. — СПб.: Питер, 2019. — 560 с.: ил. — (Серия «Библиотека программиста»).

ISBN 978-5-4461-0952-4

Как решить проблемы front-end-разработчиков и сделать их жизнь более счастливой? Нужно всего лишь познакомиться с возможностями React! Только так вы сможете быстро выйти на новый уровень и получить не только моральное, но и материальное удовлетворение от веб-разработки.

Успешные пользовательские интерфейсы должны быть визуально интересными, быстрыми и гибкими. React ускоряет тяжелые веб-приложения, улучшая поток данных между компонентами UI. Сайты начинают эффективно и плавно обновлять визуальные элементы, сводя к минимуму время на перезагрузку страниц.

Перед вами труд, над которым на протяжении полутора лет работали более дюжины человек. Тщательно отобранные примеры и подробные комментарии позволяют разработчикам перейти на React быстро, чтобы затем эффективно создавать веб-приложения, используя всю мощь JavaScript.

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

ББК 32.988.02-018
УДК 004.738.5

ISBN 978-1617293344 англ.
ISBN 978-5-4461-0952-4

Оглавление

«*React быстро* — универсальный источник информации для читателя, который желает освоить React с экосистемой инструментов, концепций и библиотек, окружающих его. Следите за объяснениями Азата, работайте над приведенными проектами — и вскоре вы начнете понимать React, Redux, GraphQL, Webpack и Jest и сможете применять их на практике».

Питер Купер (Peter Cooper), редактор «JavaScript Weekly»

«*React быстро* учит читателя наиболее ценным и заслуживающим внимания концепциям в построении современных веб-приложений с React, включая GraphQL, Webpack и прорисовку на стороне сервера. После прочтения этой книги вы будете чувствовать себя достаточно уверенно для построения веб-приложений коммерческого уровня на базе React».

Стэн Бершадский (Stan Bershadskiy), автор книги «React Native Cookbook»

«Азат — один из самых авторитетных персон в области программирования. Эта книга выходит далеко за рамки начального уровня, глубоко погружаясь в фундамент и архитектуру React. Ее стоит прочитать любому разработчику!»

Эрик Хэнчетт (Erik Hanchett), автор книги «Ember.js Cookbook»

«Стиль изложения материала достаточно прост. В книге используется очень простой язык, который поможет вам последовательно понять все рассматриваемые концепции».

Израэль Моралес (Israel Morales),
разработчик интерфейсной части и веб-дизайнер в SavvyCard

«Простой язык с несложными и логичными примерами поможет вам быстро войти в курс дела. В книге рассмотрены все основные темы, необходимые любому разработчику без опыта работы с React для того, чтобы начать писать приложения с использованием React. Авторское чувство юмора будет поддерживать ваш интерес до самых последних страниц. Я благодарен Азату за то, что он не пожалел времени и поделился с нами своим опытом React».

Сухас Дешпанде (Suhas Deshpande), программист в Capital One

«*React быстро* — отличный ресурс для изучения React. Все очень подробно и по делу. Я буду использовать эту книгу в работе над своим следующим приложением».

Натан Бэйли (Nathan Bailey),
полностековый разработчик в SpringboardAuto.com

«Азат превосходно справляется со своим делом — учить людей программированию. Книга *React быстро* содержит как фундаментальные знания, так и практические примеры, которые помогут вам быстрее взяться за программирование с использованием React».

Шу Лю (Shu Liu), IT-консультант

«С момента перехода на распространение с открытым кодом в 2013 году React.js быстро стала популярной JS-библиотекой и одним из самых высоко оцененных проектов на GitHub. В новой книге Азат Мардан в своей характерной доходчивой манере рассказывает все необходимое для изучения экосистемы React и быстрого построения высокопроизводительных SPA-приложений. Главы о состоянии React и универсальном JavaScript сами по себе оправдывают цену книги».

Пракаш Сарма (Prakash Sarma), New Star Online

«*React быстро* упростит ваш переход на React и заложит надежный фундамент для построения приложений, в полной мере реализующих все преимущества использования React».

Аллан фон Шенкель (Allan Von Schenkel), вице-президент по технологии и стратегии компании FoundHuman

«*React быстро* рассматривает все важнейшие аспекты React в простой и доступной форме. Эта книга такая же, как и все работы Азата: лаконичная и понятная, и в ней рассматривается то, что необходимо для быстрого перехода к эффективной работе. Если вы намерены добавить React в свой творческий арсенал, я рекомендую начать отсюда».

Бруно Уотт (Bruno Watt), консультант по архитектуре в hypermedia.tech

«*React быстро* — невероятно полное руководство по полностековой веб-разработке с использованием React.js. В книге рассматривается не только React, но и окружающая экосистема. Меня всегда приводило в замешательство использование React на стороне сервера; книга Азата наконец-то помогла мне разобраться в этой теме. Если у вас нет опыта работы с React, но вы хотите действительно хорошо освоить эту технологию, я бы на вашем месте остановился именно на этой книге».

Ричард Хо (Richard Kho), программист в Capital One

Посвящаю моему деду Халиту Хамитову.
Спасибо тебе за доброту и справедливость.
Ты всегда останешься в моей памяти вместе со всем,
чему ты меня научил, нашими поездками на дачу
и нашими шахматными партиями

Вступление

Я все еще надеюсь, что JavaScript умрет. Серьезно. Умрет в муках и страданиях.

Не то чтобы я ненавидел JavaScript — язык несколько улучшился за прошедшие годы. Просто я терпеть не могу лишней сложности — настолько, что назвал свой блог и бизнес «Простое программирование». Моим девизом всегда было: «Делаем сложное простым».

Сделать сложное простым нелегко. Для этого нужны особые навыки. Вы должны уметь понять сложное, и понять настолько хорошо, чтобы добраться до самой сути, — потому что на самом деле все просто. Именно это сделал Азат в своей книге.

Признаюсь, что Азату в этом немного помогли. Одна из причин, по которым лично я люблю ReactJS, — это простота. Этот фреймворк проектировался так, чтобы быть простым. Он был спроектирован для решения проблемы с ростом сложности фреймворков JavaScript и сокращения этой сложности за счет возврата к первооснове: старому доброму JavaScript. (По крайней мере, по большей части. В ReactJS существует язык JSX, который компилируется в JavaScript, но лучше вам об этом расскажет Азат.)

Хотя мне нравятся Angular, Backbone и другие фреймворки JavaScript, так как они сильно упростили создание асинхронных веб-приложений и одностраничных приложений, они также несут с собой значительную долю сложности. Шаблоны, а также понимание синтаксиса и всех нюансов этих фреймворков делают работу более эффективной, но фактически сложность при этом выводится изнутри наружу. ReactJS действует иначе, избавляется от шаблонов и предоставляет средства для применения компонентной архитектуры в пользовательском интерфейсе с использованием JavaScript. Мне нравится такой подход. Он прост. Но даже самые простые вещи бывает сложно объяснить — или, что еще хуже, они становятся сложными в изложении преподавателя.

И здесь вступает в дело Азат. Он умеет объяснять. Умеет упрощать. В начале книги автор представляет React, сравнивая его с тем, что вам, скорее всего, уже известно — с Angular. Впрочем, даже если вы не знаете Angular, его объяснение ReactJS быстро поможет вам понять азы и предназначение этого продукта. Затем Азат быстро показывает, как создать простое приложение ReactJS, чтобы вы могли все увидеть и самостоятельно проверить на практике. После этого он излагает оставшиеся 20 % того, что необходимо знать для выполнения 80 % повседневной работы

в React; при этом он использует реальные примеры, понятные каждому читателю. Наконец — и это моя любимая часть книги! — он приводит многочисленные примеры и учебные проекты. Безусловно, лучший способ узнать что-либо — практика. Под руководством Азата вы пройдете через создание шести (да, шести!) нетривиальных проектов с использованием ReactJS.

А теперь, в полном соответствии со своей любимой темой — простотой, я завершаю свое вступление. Скажу лишь, что эта книга — попросту лучший известный мне способ изучения ReactJS.

Джон Сонмез (John Sonmez)
Автор книги «Soft Skills» (http://amzn.to/2hFHXAu)
и основатель «Simple Programmer» (https://simpleprogrammer.com)

Предисловие

Шел 2008 год, банки закрывались сплошь и рядом. Я работал в Федеральной корпорации страхования депозитов (FDIC), главной задачей которой была компенсация вкладов закрывшихся и неплатежеспособных банков. Признаюсь, в отношении гарантии занятости моя должность была где-то на уровне работы в «Леман Бразерс» или продажи билетов на «Титаник». Но пока времена сокращения бюджета для моего отдела еще были в далеком будущем, мне представилась возможность поработать над приложением, которое называлось «электронным оценщиком депозитной страховки», или EDIE (Electronic Deposit Insurance Estimator). Приложение пользовалось огромной популярностью по простой причине: людям хотелось узнать, какая часть их сбережений была застрахована федеральным правительством Соединенных Штатов, и программа EDIE оценивала эту величину.

Однако здесь скрывалась одна загвоздка: люди не желают сообщать правительству о своих частных счетах. Для защиты конфиденциальности приложение было построено полностью из JavaScript, HTML и CSS клиентской части без каких-либо технологий серверной части. Таким образом, FDIC не собирала никакой финансовой информации.

Приложение представляло собой месиво из «спагетти-кода», который сформировался после десятков итераций, проведенных с консультантами. Разработчики приходили и уходили, не оставляя после себя ни документации, ни чего-либо напоминающего логичные, простые алгоритмы. Разбираясь в коде, я чувствовал себя как турист, пытающийся воспользоваться нью-йоркским метро без карты. Там были мириады функций, вызывающих другие функции, странные структуры данных и новые функции... В современной терминологии это было чистое UI-приложение, потому что у него не было серверной части с хранением данных.

Как жаль, что у меня тогда не было React.js. С React приятно работать. Это новый менталитет — новый подход к разработке. Простота хранения всей базовой функциональности в одном месте, в отличие от разбиения ее на HTML и JS, снимает ограничения. Она разожгла мой энтузиазм к разработке клиентской части.

React — свежий взгляд на разработку компонентов пользовательского интерфейса. Это новое поколение библиотек презентационного уровня. В сочетании с моделью и библиотекой маршрутизации React может заменить Angular, Backbone или Ember в технологическом стеке веб-программирования и мобильной разработки. Как раз

по этой причине я написал эту книгу. Angular мне никогда не нравился: он слишком сложен и устанавливает слишком жесткие ограничения. Шаблонизатор сильно привязан к предметной области, до такой степени, что это уже не JavaScript, а другой язык. Я использовал Backbone.js; этот фреймворк нравится мне своей простотой и подходом «сделай сам». Он прошел проверку временем и больше напоминает основу для создания собственных фреймворков, нежели полноценный фреймворк. Проблема с Backbone — повышение сложности взаимодействий между моделями и представлениями; разные представления обновляют разные модели, которые обновляют другие представления, инициирующие события моделей.

Мой личный опыт проведения на Kickstarter кампании по созданию учебного курса React.js (http://mng.bz/XgkO) и посещения различных конференций и мероприятий показывает, что разработчики всегда ищут более эффективные способы разработки пользовательских интерфейсов. Большая часть коммерческой ценности в наши дни лежит в пользовательских интерфейсах, серверная часть — всего лишь ресурс. В области залива Сан-Франциско, где я живу и работаю, большинство вакансий программистов открывается для разработчиков интерфейсов или (модный новый термин) разработчиков широкого профиля/полностековых разработчиков. Лишь в нескольких больших компаниях, таких как Google, Amazon и Capital One, существует достаточно сильный спрос на специалистов по теории данных и программистов серверной части.

Лучший способ сохранить рабочее место или найти хорошую вакансию — стать разработчиком широкого профиля (generalist). А для этого проще всего использовать изоморфную, масштабируемую, удобную для разработчика библиотеку (такую, как React) в клиентской части — в сочетании с Node.js для серверной части на случай, если вам все же придется возиться с кодом на стороне сервера.

Для мобильных разработчиков HTML5 еще два или три года назад считался неприличным словом. Facebook отказался от своего приложения HTML5 в пользу более производительной платформенной реализации. Тем не менее это неблагоприятное отношение быстро изменяется. С React Native можно генерировать разметку для мобильных приложений; вы можете оставить свои компоненты пользовательского интерфейса, но адаптировать их к другой среде — еще один довод в пользу изучения React.

Программирование может быть творческим делом. Вы не отвлекаетесь на рутинные задачи, сложность и мнимое разделение обязанностей. Вы отсекаете все лишнее и *направляете свои творческие силы на упрощенную красоту модульных компонентных пользовательских интерфейсов на базе React.* Добавьте Node для изоморфного/универсального JavaScript, и вы постигнете дзен.

Приятного чтения, и расскажите мне, понравилась ли вам книга, — оставьте отзыв на Amazon.com (http://amzn.to/2gPxv9Q).

Благодарности

Я хочу поблагодарить интернет, Вселенную и человеческую изобретательность, благодаря которой стала возможна телепатия. Не открывая рта, я могу поделиться своими мыслями с миллионами людей по всему миру в социальных сетях, таких как Twitter, Facebook и Instagram. Ура!

Я беспредельно благодарен своим учителям — как тем, которые целенаправленно учили меня в школах и университетах, так и случайным, которые поделились своей мудростью в книгах и в общении.

Как однажды написал Стивен Кинг: «Писать — удел человеческий, редактировать — удел божественный». А значит, я бесконечно благодарен редакторам этой книги и в еще большей степени — читателям, которые столкнутся с неизбежными опечатками и ошибками. Это моя четырнадцатая книга, и я знаю, что *очепятки* все равно будут, несмотря на все мои старания.

Спасибо персоналу издательства Manning, благодаря которому эта книга стала возможной; издателю Маржану Бейсу (Marjan Bace) и всем участникам редакторской и производственной групп, включая Дженет Вейл (Janet Vail), Кевина Салливана (Kevin Sullivan), Тиффани Тейлор (Tiffany Taylor), Кэти Теннант (Katie Tennant), Джордана Салиновича (Gordan Salinovic), Дэна Махарри (Dan Maharry) и многих других, остававшихся в тени.

Мне не хватает слов для выражения благодарности потрясающей группе научных редакторов под руководством Айвена Мартиновича (Ivan Martinovic): Джеймсу Анайпакосу (James Anaipakos), Дейну Балье (Dane Balia), Арту Бергквисту (Art Bergquist), Джоэлю Голдфингеру (Joel Goldfinger), Питеру Хэмптону (Peter Hampton), Луису Мэттью Хеку (Luis Matthew Heck), Рубену Дж. Леону (Ruben J. Leon), Джеральду Мэку (Gerald Mack), Камалю Раджу (Kamal Raj) и Лукасу Тетаманти (Lucas Tettamanti). В частности, они обнаруживали технические ошибки, ошибки в терминологии и опечатки, а также предлагали темы для рассмотрения. Каждая итерация процесса редактирования, каждое изменение, реализованное на основании обсуждений на форуме, способствовали формированию текста книги.

Я хочу особо поблагодарить Анто Аравинта (Anto Aravinth), занимавшегося техническим редактированием, и Германа Фриджерио (German Frigerio), который был корректором. Это лучшие технические специалисты, на которых я только мог надеяться.

Огромное спасибо Джону Сонмезу (John Sonmez) из компании Pluralsight, издательства Manning и SimpleProgrammer.com, за предисловие к книге. Спасибо вам, Питер Купер (Peter Cooper), Эрик Хэнчетт (Erik Hanchett) и Стэн Бершадский (Stan Bershadskiy), за обзоры и большую достоверность материала. Читателям, которые никогда не слышали о Джоне, Питере, Эрике или Стэне, стоит подписаться на их публикации и понаблюдать за их работой в области разработки.

Наконец, спасибо всем участникам программы MEAP за обратную связь. Переработка книги на основании ваших отзывов задержала публикацию на год, но в результате появилась лучшая книга о React из всех опубликованных на сегодняшний день.

О книге

Эта книга была написана для того, чтобы помочь разработчикам решить некоторые проблемы, сделать их жизнь более содержательной и счастливой, а также помочь им больше зарабатывать благодаря знакомству с React.js — и притом быстро. Над этой книгой работало более десятка людей в течение полутора лет. Как минимум эта книга должна подготовить читателя к таким необычным концепциям, как JSX, однонаправленная передача данных и декларативное программирование.

Структура книги

Книга делится на две части: «Основы React» (главы 1–11) и «Архитектура React» (главы 12–20). В каждой главе приводится текстовое описание, дополненное примерами кода и диаграммами, где это уместно. Каждая глава написана как самостоятельная; это означает, что у вас не должно быть проблем с чтением книги не по порядку глав — хотя я рекомендую читать последовательно. В конце каждой главы приводятся вопросы, закрепляющие понимание материала, и краткая сводка.

Каждая часть завершается несколькими большими проектами, которые расширят ваш опыт работы с React и укрепят понимание темы как логического продолжения концепций и материалов предыдущих глав. Эти проекты являются неотъемлемой частью книги — старайтесь не пропускать их.

Я советую вводить каждую строку кода самостоятельно, не прибегая к копированию/вставке. Исследования показали, что письменный ввод и набор на клавиатуре повышает эффективность обучения. Книга завершается пятью приложениями со вспомогательным материалом. Ознакомьтесь с ними (а также прочитайте оглавление) перед тем, как начинать чтение.

Веб-сайты этой книги — www.manning.com/books/react-quickly и http://reactquickly.co. Если вам понадобится новейшая информация, скорее всего, вы найдете ее здесь.

Исходный код доступен на сайте Manning (www.manning.com/books/react-quickly) и GitHub (https://github.com/azat-co/react-quickly). За дополнительной информацией обращайтесь к разделу «Исходный код». В книге приводятся полные листинги — это намного удобнее, чем переходить к GitHub или редактору кода для просмотра файлов.

Для кого написана эта книга (обязательно прочитайте!)

Эта книга написана для веб-программистов, мобильных разработчиков и программистов с двух-трехлетним опытом работы, которые хотят изучить и использовать React.js для веб-программирования и мобильной разработки. Можно сказать, что книга написана для людей, которые знают наизусть комбинацию клавиш для Developer Tools (Cmd+Opt+J или Cmd+Opt+I на Mac). Книга предназначена для читателей, которые знакомы со следующими концепциями накоротке:

- Одностраничные приложения (SPA).
- REST-совместимые службы и архитектура API.
- JavaScript и особенно замыкания, области видимости, методы строк и массивов.
- HTML, HTML5, их элементы и атрибуты.
- CSS, стили и селекторы JavaScript.

Опыт работы с jQuery, Angular, Ember.js, Backbone.js или других фреймворков в стиле MVC пригодится, потому что вы сможете сравнить их с подходом React. Тем не менее это не необходимо и до какой-то степени может быть вредно, потому что вам придется забывать некоторые паттерны — React не следует некоторым канонам MVC.

Мы будем использовать инструменты командной строки, так что если вы боитесь их, самое время побороть свою фобию командной строки/терминала. Как правило, средства командной строки (CLI) превосходят по мощи и гибкости свои версии с графическим интерфейсом (сравните командную строку Git с настольной версией GitHub — в последней я до сих пор не разобрался).

Определенное сходство с Node.js позволит вам изучить React намного быстрее, чем читателю, который никогда не слышал о Node.js, npm, Browserify, CommonJS, Gulp или Express.js. Я написал несколько книг о Node.js для тех, кто захочет освежить свои знания; самая популярная из них — «Practical Node.js» (http://practicalnodebook.com). Также вы можете пройти в интернете бесплатное приключение NodeSchool (http://nodeschool.io) («бесплатно» не всегда означает «хуже»).

Кому эта книга не подойдет (тоже прочитайте!)

Эта книга не является исчерпывающим справочником по веб-программированию и мобильной разработке. Предполагается, что вы уже разбираетесь в этих областях. Если вам нужна информация о базовых концепциях программирования или

основах JavaScript, на эти темы написано немало хороших книг, поэтому я не буду повторять уже существующие материалы в этой книге.

Как использовать эту книгу

Прежде всего книгу нужно *прочитать*. Это не шутка — многие люди покупают книги, но не читают их. Обычно подобное случается с цифровыми копиями, потому что они хранятся где-то на диске или в облаке и о них забывают. Прочитайте книгу и проработайте проекты, главу за главой.

В каждой главе рассматривается определенная тема или группа тем, логически вытекающих одна из другой. По этой причине я рекомендую прочитать эту книгу от начала к концу, а затем возвращаться к отдельным главам за информацией. Но как я уже говорил выше, вы также можете читать отдельные главы в произвольном порядке, потому что проекты глав не зависят друг от друга.

В тексте встречаются многочисленные ссылки на внешние ресурсы. Большинство из них не являются обязательными, но они предоставляют дополнительную информацию по теме. Я рекомендую читать книгу поблизости от компьютера, чтобы открывать ссылки в тексте.

Часть текста оформлена моноширинным шрифтом: `getAccounts()`. Это означает, что этот текст содержит код (отдельные конструкции и целые блоки). Иногда встречается код с необычными отступами:

```
      document.getElementById('end-of-time').play()
    }
```

Это означает, что я описываю большой фрагмент кода и разбил его на части. Этот код принадлежит более крупному листингу, начинающемуся с позиции `0`; сам по себе этот маленький фрагмент работать не будет.

В других случаях блоки кода не имеют отступа. Тогда можно с уверенностью считать, что этот блок самостоятелен:

```
ReactDOM.render(
<Content />,
  document.getElementById('content')
)
```

Если в тексте встречается знак `$`, перед вами команда терминала/командной строки. Пример:

```
$ npm install -g babel@5.8.34
```

Самое важное, что нужно знать и помнить во время чтения книги: вам должно быть интересно. Если вам неинтересно, это не JavaScript!

Исходный код

Весь код этой книги доступен по адресам www.manning.com/books/react-quickly и https://github.com/azat-co/react-quickly. В нем используется соглашение об именах папок ch*NN*, где *NN* — номер главы с начальным 0 при необходимости (например, ch02 для кода главы 2). Исходный код в репозитории GitHub будет дорабатываться: в него будут включаться патчи, исправления ошибок, а возможно, даже новые версии и стили (ES2020?).

Об авторе

Я опубликовал более 14 книг и 17 сетевых учебных курсов (https://node.university), в основном доступных на облачных платформах, по темам React,JavaScript и Node.js. (Одна книга написана о том, как писать книги, а другая — о том, что делать после написания нескольких книг.)

Прежде чем сосредоточиться на Node, я программировал на других языках (Java, C, Perl, PHP, Ruby) практически все время после окончания средней школы (более 12 лет); на это у меня ушло определенно более рекомендованных 10 000 часов[1].

В настоящее время я работаю техническим директором в одном из десяти крупнейших банков США, также входящих в список Fortune 500: Capital One Financial Corporation в прекрасном Сан-Франциско. До этого я работал в небольших стартапах, гигантских корпорациях и даже в федеральном правительстве США, занимаясь разработкой настольных, мобильных и веб-приложений, преподаванием, распространением знаний в области разработки и управления проектами.

Не стану тратить слишком много вашего времени на рассказы о себе; вы можете узнать больше в моем блоге (http://webapplog.com/about) и в социальных сетях (www.linkedin.com/in/azatm). Вместо этого я хочу рассказать о моем опыте в той области, которая имеет прямое отношение к книге.

Когда я переехал в солнечный штат Калифорния в 2011 году, чтобы поступить в стартап и пройти программу бизнес-акселератора[2] (если вам интересно, это была программа 500 Startups), я начал использовать современный JavaScript. Я изучил Backbone.js, чтобы построить несколько приложений для стартапа, и эта работа произвела на меня впечатление. Фреймворк был огромным шагом вперед в организации кода по сравнению с другими одностраничными приложениями, которые я строил до этого. В нем были маршруты и модели. Вот это да!

Мне представилась еще одна возможность увидеть выдающуюся силу Backbone и изоморфного JavaScript, когда я работал руководителем группы программистов

---

[1] См. книгу М. Гладуэлла «Гении и аутсайдеры» (https://en.wikipedia.org/wiki/Outliers_(book)).

[2] https://ru.wikipedia.org/wiki/Бизнес-акселератор.

в компании DocuSign, своего рода Google в мире электронных подписей (ее доля рынка составляет 70 %). Мы переработали монолитное веб-приложение ASP.NET семилетней давности, у которого выпуск дополнительной версии занимал четыре недели, в компактное приложение Backbone-Node-CoffeeScript-Express с отличным пользовательским интерфейсом и выпуском дополнительной версии всего за одну-две недели. Дизайнеры хорошо поработали над удобством использования. Не стоит и говорить, что в приложении было великое множество UI-представлений с различными степенями интерактивности.

Конечное приложение было изоморфным еще до того, как появился такой термин. Мы использовали модели Backbone на сервере для предварительной выборки данных из API и их кэширования. Одни и те же шаблоны Jade использовались в браузере и на сервере.

Это был интересный проект, который еще сильнее убедил меня в мощи использования одного языка во всем стеке. Разработчики с опытом работы на C# и интерфейсном JavaScript (в основном jQuery), работавшие над старым приложением, проводили спринт (один цикл выпуска — обычно неделя или две) и влюблялись в четкость структуры CoffeeScript, организацию Backbone и скорость Node (как в разработке, так и в плане скорости выполнения).

За десятилетие в веб-разработке я сталкивался с хорошими, плохими и уродливыми (в основном уродливыми) сторонами интерфейсной разработки. Как оказалось, нет худа без добра, потому что я начал еще больше ценить React после перехода.

Если вы хотите получать обновления, новости и полезные советы, свяжитесь со мной в Сети: подпишитесь, возьмите в друзья, просто читайте — выбирайте сами:

- Twitter — https://twitter.com/azatmardan
- Website — http://azat.co
- LinkedIn — http://linkedin.com/in/azatm
- Блог — http://webapplog.com
- Publications — http://webapplog.com/books

Об обложке

Один из читателей издания прислал мне сообщение с вопросом о дервише на обложке. Да, этого персонажа можно легко принять за перса или представителя одного из кочевых тюркских племен, населявших Ближний Восток и Центральную Азию. Отчасти это обусловлено высоким уровнем развития торговли и путешествий в тех регионах на протяжении многих веков. Но по словам художника, нарисовавшего эту картину, он рисовал сибирского башкира. Большинство современных башкиров живет в Республике Башкортостан (или Башкирия). Башкиры являются близким

этническими и географическими соседями волжских булгар (ошибочно называемых татарами): башкиры и татары являются вторым по численности народом Российской Федерации (на первом месте русские, если вас интересует).

Иллюстрация позаимствована с гравюры XVIII века «Gravure Homme Baschkir» Жака Грассе де Сен-Совёр (Jacques Grasset de Saint-Sauveur). Увлечение дальними странами и путешествиями для удовольствия в то время только начиналось, и коллекции подобных рисунков были популярны; они знакомили как путешественников, так и домоседов, черпавших информацию из книг и картин, с жителями других стран. Разнообразие рисунков ярко напоминает нам, насколько непохожими в культурном отношении были города и области мира всего 200 лет назад. Люди, жившие в изоляции друг от друга, говорили на разных диалектах и языках. На улицах городов и в сельской местности можно было легко узнать, где живут люди и чем они занимаются, — достаточно было взглянуть на их одежду.

С тех пор стандарты внешнего вида изменились, и региональные различия, столь богатые в то время, остались в прошлом. Сейчас уже трудно отличить друг от друга жителей разных континентов, не говоря уже о разных городах или областях. Возможно, мы пожертвовали культурным разнообразием ради более разнообразной личной жизни — и безусловно, ради более разнообразной и стремительной технологической жизни.

Сейчас, когда все компьютерные книги похожи друг на друга, издательство Manning стремится к разнообразию и помещает на обложки книг иллюстрации, показывающие разнообразие жизни двухсотлетней давности.

От издательства

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

Мы будем рады узнать ваше мнение!

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

В начале каждой главы вставлены ссылки на авторские видеоролики. Обращаем ваше внимание, что ролики доступны на английском языке.

ЧАСТЬ 1

Основы React

Привет! Меня зовут Азат Мардан, и я собираюсь отправиться с вами в путешествие по удивительному миру React. React сделает программирование клиентской части более творческим, упростит написание и сопровождение кода, а пользователи будут поражены скоростью работы ваших веб-приложений. Появление React в корне изменило обстановку в веб-программировании: сообщество React положило начало многих подходов, понятий и паттернов проектирования, а другие библиотеки следовали по уже проложенному пути.

Я преподавал этот материал более 20 раз на своих сетевых уроках и обычных учебных курсах сотням программистов с совершенно разными уровнями квалификации и практического опыта. Таким образом, этот материал прошел проверку на моих студентах; вы получите переработанную и наиболее эффективную версию моего курса основ React в письменном виде. Главы этой части очень важны для вашего знакомства с React.

Главы 1–11 являются результатом почти двухлетней работы нескольких человек, но они читаются как последовательность тем, развивающих друг друга. Лучше всего начать с главы 1 и читать последовательно. Каждая глава включает небольшое видеопослание от меня; главы 1–8 завершаются упражнениями, а главы 9–11, которые представляют собой проекты, содержат упражнения для самостоятельной работы.

В целом эта часть книги закладывает прочную основу для изучения концепций, паттернтов и возможностей React. Сможете ли вы приехать в незнакомую страну и понять чужой язык, который вы не изучали? Нет, и именно поэтому необходимо изучить «язык» React, прежде чем пытаться строить сложные приложения. Следовательно, для вас очень важно изучить эти базовые концепции React — чтобы вы изучили язык React; именно этим мы и будем заниматься в следующих 11 главах.

Итак, пора начинать знакомство с React — и учиться бегло говорить «на языке React».

1 Знакомство с React

Посмотрите вступительный видеоролик к этой главе, отсканировав QR-код или перейдя по ссылке http://reactquickly.co/videos/ch01[1].

ЭТА ГЛАВА ОХВАТЫВАЕТ СЛЕДУЮЩИЕ ТЕМЫ:

- Понимание того, что такое React.
- Решение проблем с помощью React.
- Установка React в веб-приложения.
- Написание первого приложения React: Hello World.

Когда я только начинал работать в области веб-разработки в начале 2000 года, все, что мне было нужно — HTML и серверный язык (такой, как Perl или PHP). Старые добрые времена, когда для обычной отладки кода клиентской части (frontend) приходилось набивать его вызовами `alert()`! Время шло, интернет-технологии развивались, и сложность построения сайтов радикально возросла. Сайты превратились в веб-приложения со сложными интерфейсами, бизнес-логикой и уровнями данных, которые требовали постоянных изменений и обновлений — часто в реальном времени.

Для решения проблем с построением сложных пользовательских интерфейсов (UI, User Interface) было написано много библиотек шаблонов JavaScript. Однако все они требовали, чтобы разработчики придерживались классического «разделения труда» — то есть разделения стилевого оформления (CSS), данных и структуры (HTML) и динамических взаимодействий (JavaScript) — и не удовлетворяли современных потребностей. (Помните термин *DHTML*?)

С другой стороны, React предлагает новый подход, упрощающий разработку клиентской части. React — мощная библиотека пользовательского интерфейса — предоставляет альтернативу, которая была принята многими крупными компаниями, такими

[1] Обращаем ваше внимание, что ролики доступны на английском языке.

как Facebook, Netflix и Airbnb, и рассматривается ими как перспективное решение. Вместо создания одноразовых шаблонов для ваших пользовательских интерфейсов React позволяет создавать на JavaScript UI-компоненты, предназначенные для повторного использования, которые можно снова и снова использовать в ваших сайтах.

Вам нужен элемент для вывода контрольного изображения (captcha) или выбора даты? Воспользуйтесь React для определения компонента `<Captcha />` или `<DatePicker />`, который можно добавить к форме: простой подключаемый компонент, содержащий всю функциональность и логику для взаимодействия с серверной частью. Понадобилось поле с автозаполнением, которое обращается с асинхронным запросом к базе данных после того, как пользователь введет четыре и более буквы? Определите компонент `<Autocomplete charNum="4"/>` для асинхронного запроса. Вы даже можете выбрать, будет ли такое поле иметь интерфейс текстового поля или же вместо него будет использоваться другой нестандартный элемент формы — возможно, `<Autocomplete textbox="..." />`.

Такой подход не нов. Идея построения пользовательских интерфейсов из компонентов появилась давно, но библиотека React стала первой, которая предоставляла такую возможность с использованием чистого JavaScript без шаблонов. И такой подход оказался более простым в сопровождении, повторном использовании и расширении.

React — превосходная библиотека для построения пользовательских интерфейсов, и она должна стать частью вашего веб-инструментария. В этой главе рассматриваются плюсы и минусы использования React в приложениях и возможность интеграции React в существующий стек веб-разработки.

Часть I посвящена основным концепциям и возможностям React, а в части II рассматривается работа с библиотеками, связанными с React, для построения более сложных приложений клиентской части (то есть «стек React»). В каждой части продемонстрирована как разработка с нуля, так и разработка с унаследованным кодом и существующими системами; таким образом, вы получите представление о том, как применять стек React в реальных ситуациях.

ВИДЕОРОЛИКИ И ИСХОДНЫЙ КОД

Все учатся по-разному. Одни лучше воспринимают текст, другие — видео, третьи — личное общение. Каждая глава книги включает короткий видеоролик, в котором суть главы объясняется менее чем за 5 минут. Смотреть ролики совершенно не обязательно. Они содержат сводку на тот случай, если вы предпочитаете видеоформат или захотите освежить память. После каждого ролика можно решить, продолжать ли читать эту главу или перейти к следующей.

Исходный код примеров этой главы доступен по адресам www.manning.com/books/react-quickly и https://github.com/azat-co/react-quickly/tree/master/ch01 (папка ch01 репозитория GitHub https://github.com/azat-co/react-quickly). Некоторые демонстрационные примеры также доступны по адресу http://reactquickly.co/demos.

1.1. Что такое React?

Чтобы правильно представить React.js читателю, необходимо начать с определения. Итак, что же такое React? Это библиотека компонентов пользовательского интерфейса (UI-компонентов). UI-компоненты создаются с React на языке JavaScript, а не на специальном языке шаблонов. Этот подход, называемый *созданием составного пользовательского интерфейса*, играет фундаментальную роль в философии React.

UI-компоненты React представляют собой автономные блоки функциональности, предназначенные для конкретной цели. Например, можно представить себе компоненты для выбора даты, контрольного изображения, ввода адреса или почтового индекса. Такие компоненты обладают как визуальным представлением, так и динамической логикой. Некоторые компоненты даже способны на самостоятельные взаимодействия с сервером: например, компонент автозаполнения может загружать список с сервера.

ПОЛЬЗОВАТЕЛЬСКИЕ ИНТЕРФЕЙСЫ

В широком смысле к пользовательскому интерфейсу[1] относится всё, что обеспечивает взаимодействие между компьютером и человеком. Представьте перфокарту или мышь: и то и другое относится к пользовательскому интерфейсу. В том, что касается ПО, программисты говорят о графических пользовательских интерфейсах, или просто графических интерфейсах (GUI, Graphic User Interface), которые разрабатывались для ранних моделей персональных компьютеров, таких как Mac и PC. Графический интерфейс состоит из меню, текста, значков, изображений, рамок и других элементов. Веб-элементы образуют узкое подмножество графического интерфейса: они существуют в браузерах, но также есть элементы для настольных приложений Windows, OS X и других операционных систем.

Каждый раз, когда в книге упоминается пользовательский интерфейс(UI), я имею в виду графический веб-интерфейс (GUI).

Компонентная архитектура, или *CBA* (Component-Based Architecture), — не стоит путать с веб-компонентами, которые составляют всего лишь одну из последних реализаций CBA, — существовали еще до появления React. Такие архитектуры обычно создают меньше проблем с повторным использованием, сопровождением и расширением, в отличие от монолитных пользовательских интерфейсов. React добавляет в эту картину использование чистого JavaScript (без шаблонов) и новый подход к формированию компонентов.

[1] https://ru.wikipedia.org/wiki/Интерфейс_пользователя.

1.2. Проблема, которую решает React

Какую проблему решает React? В контексте нескольких последних лет веб-разработки обращают на себя внимание проблемы с построением и управлением сложными веб-интерфейсами для приложений клиентской части: библиотека React строилась прежде всего для решения этих проблем. Представьте себе большое веб-приложение, такое как Facebook: одной из самых неприятных задач при разработке таких приложений становится управление изменениями представлений в ответ на изменения в данных.

Дополнительную информацию о задачах, которые помогает решать React, можно найти на официальном сайте React: «Мы строили React для решения одной проблемы: построения больших приложений с данными, изменяющимися во времени»[1].

Интересно! Еще больше информации можно найти в истории React. В обсуждении из подкаста React[2] упоминается, что создатель React — Джордан Уок (Jordan Walke) — занимался решением задачи для Facebook: обновлением поля с автозаполнением по нескольким источникам данных. Данные поступали асинхронно с серверной части. Становилось все сложнее определять, куда вставлять новые строки, чтобы повторно использовать элементы DOM. Уок решил каждый раз заново генерировать представление поля (элементы DOM). Такое решение было элегантным в своей простоте: пользовательский интерфейс в виде функций. Вызывая их с данными, можно было получить предсказуемое отображение представлений.

Позднее оказалось, что генерирование элементов в памяти происходит в высшей степени быстро, а «узким местом» является рендеринг в DOM. Однако команда React предложила алгоритм, избегающий ненужной возни с DOM. В результате библиотека React стала работать очень быстро (и не требовала многого в отношении затрат ресурсов). Превосходная эффективность, удобство для разработчика и компонентная архитектура обусловили успех React. Эти и другие преимущества React описаны в следующем разделе.

Библиотека React решила исходную проблему Facebook, и многие крупные фирмы согласились с этим подходом. React применяется достаточно широко, а популярность библиотеки растет с каждым месяцем. Технология React родилась в Facebook[3], а сейчас ее использует не только Facebook, но и Instagram, PayPal, Uber, Сбербанк,

---

[1] Официальный веб-сайт React, «Why React?», 24 марта 2016 г., http://bit.ly/2mdCJKM.

[2] React Podcast, «8. React, GraphQL, Immutable & Bow-Ties with Special Guest Lee Byron», 31 декабря 2015 г., http://mng.bz/W1X6.

[3] «Introduction to React.js», 8 июля 2013 г., http://mng.bz/86XF.

Asana[1], Khan Academy[2], HipChat[3], Flipboard[4] и Atom[5] — и это далеко не полный список[6]. Многие из этих приложений изначально использовали другую технологию (обычно шаблонизаторы с Angular или Backbone), но перешли на React и были в высшей степени довольны.

1.3. Преимущества React

Создатели каждой новой библиотеки или фреймворка утверждают, что их детище в каком-то отношении превосходит своих предшественников. Вначале была библиотека jQuery, и она стала огромным шагом вперед для написания межбраузерного кода на чистом JavaScript. Если вы помните, с учетом особенностей Internet Explorer и WebKit-браузеров один вызов AJAX мог занимать много строк кода. С jQuery же все ограничивалось одним-единственным вызовом: `$.ajax()`. Тогда jQuery называли фреймворком — но не сейчас! В наши дни фреймворком считается нечто более масштабное и мощное.

Аналогичным образом дело обстояло с Backbone, а затем и с Angular; каждое новое поколение фреймворков JavaScript приносило что-то новое. В этом React не уникален. Новым было то, что React ниспровергает некоторые базовые концепции, используемые самыми популярными фреймворками: например, идею о необходимости шаблонов.

Ниже перечислены некоторые преимущества React перед другими библиотеками и фреймворками:

- *Простые приложения* — React использует компонентную архитектуру с чистым JavaScript, декларативный стиль программирования и мощные, удобные для разработчика абстракции DOM (и не только DOM, но и iOS, Android и т. д.).
- *Быстрый пользовательский интерфейс* — React обеспечивает выдающееся быстродействие благодаря своей виртуальной модели DOM и алгоритму интеллектуального согласования (smart-reconciliation algorithm). Одно из побочных преимуществ — возможность тестирования без запуска браузера с отсутствующим графическим интерфейсом.

---

[1] Malcolm Handley and Phips Peter, «Why Asana Is Switching to TypeScript», Asana Blog, November 14, 2014, http://mng.bz/zXKo.

[2] Joel Burget, «Backbone to React», http://mng.bz/WGEQ.

[3] Rich Manalang, «Rebuilding HipChat with React.js», Atlassian Developers, 10 февраля 2015 г., http://mng.bz/r0w6.

[4] Michael Johnston, «60 FPS on the Mobile Web», Flipboard, 10 февраля 2015 г., http://mng.bz/N5F0.

[5] Nathan Sobo, «Moving Atom to React», Atom, 2 июля 2014 г., http://mng.bz/K94N.

[6] Также см. статистику использования JavaScript по адресу http://libscore.com/#React.

- *Сокращение объема кода* — огромное сообщество React и гигантская экосистема компонентов предоставляют в распоряжение разработчика множество разнообразных библиотек и компонентов. Это важный момент при выборе фреймворка, используемого для разработки.

React обладает многими возможностями, которые упрощают работу по сравнению с другими фреймворками для построения клиентской части. Разберем все эти пункты подробнее, один за другим.

1.3.1. Простота

Концепция простоты в компьютерных технологиях высоко ценится как разработчиками, так и пользователями. Речь идет не о простоте использования — простое решение может быть более сложным в реализации, но в конечном итоге оно оказывается более элегантным и эффективным, а простое на первый взгляд нередко оказывается сложным. Простота тесно связана с принципом KISS (Keep It Simple, Stupid, то есть «не усложняй»[1]). Суть в том, что простые системы лучше работают.

Подход React позволяет строить более простые решения с применением кардинально улучшенного процесса веб-разработки. Когда я начинал работать с React, это стало для меня существенным шагом в положительном направлении, который я бы мог сравнить с переходом от простого JavaScript без фреймворков на jQuery.

В React эта простота достигается благодаря следующим особенностям:

- *Декларативный стиль* (в отличие от императивного) — React отдает предпочтение декларативному стилю вместо императивного, автоматически обновляя представления.
- *Компонентная архитектура, использующая чистый JavaScript,* — React не использует для своих компонентов *предметно-ориентированные языки* (DSL, Domain-Specific Languages), только чистый JavaScript. Здесь нет разделения, когда работаешь над одной и той же функциональностью.
- *Мощные абстракции* — в React есть упрощенный механизм реализации в DOM, который позволяет нормализовать обработку событий и другие интерфейсы, работающие одинаково в разных браузерах.

Рассмотрим последовательно все эти пункты.

Декларативный стиль

Прежде всего React выбирает декларативный стиль вместо императивного. Декларативный стиль означает, что разработчик определяет, *что должно получиться*, а не *как это делать* шаг за шагом. Но почему декларативный стиль лучше? Его главное преимущество — уменьшение сложности, упрощение чтения и понимания кода.

[1] https://ru.wikipedia.org/wiki/KISS_(принцип).

Следующий короткий пример на JavaScript демонстрирует различия между декларативным и императивным программированием. Допустим, требуется создать массив (`arr2`), элементы которого вычисляются удвоением элементов другого массива (`arr`). Вы можете в цикле `for` перебрать элементы массива и приказать системе умножить каждый элемент на 2 и создать новый элемент (`arr2[i]=`):

```
var arr = [1, 2, 3, 4, 5],
  arr2 = []
for (var i=0; i<arr.length; i++) {
  arr2[i] = arr[i]*2
}
console.log('a', arr2)
```

Полученный в результате удвоения элементов массив выводится в консоль:

```
a [2, 4, 6, 8, 10]
```

Это пример императивного программирования, и оно работает, а потом в какой-то момент перестает работать из-за сложности кода. При слишком большом количестве императивных команд разработчику становится сложнее понять, как должен выглядеть конечный результат. К счастью, ту же логику можно переписать в декларативном стиле с `map()`:

```
var arr = [1, 2, 3, 4, 5],
  arr2 = arr.map(function(v, i){ return v*2 })
console.log('b', arr2)
```

Фрагмент выводит `b [2, 4, 6, 8, 10]`; переменная `arr2` содержит те же элементы, что и в прошлом примере. Какой фрагмент кода проще прочесть и понять? По моему скромному мнению, декларативный.

Взгляните на следующий императивный код для получения вложенного значения объекта. Выражение должно вернуть значение на основе строки (например, `account` или `account.number`) так, чтобы следующие команды выводили `true`:

```
var profile = {account: '47574416'}
var profileDeep = {account: { number: 47574416 }}
console.log(getNestedValueImperatively(profile, 'account') === '47574416')
console.log(getNestedValueImperatively(profileDeep, 'account.number')
➥ === 47574416)
```

Императивный стиль буквально сообщает системе, что следует делать для достижения нужных результатов:

```
var getNestedValueImperatively = function getNestedValueImperatively
➥ (object, propertyName) {
  var currentObject = object
  var propertyNamesList = propertyName.split('.')
  var maxNestedLevel = propertyNamesList.length
  var currentNestedLevel
```

```
  for (currentNestedLevel = 0; currentNestedLevel < maxNestedLevel;
  ➥ currentNestedLevel++) {
     if (!currentObject || typeof currentObject === 'undefined')
     ➥  return undefined
     currentObject = currentObject[propertyNamesList[currentNestedLevel]]
  }

  return currentObject
}
```

Сравните с декларативным стилем (ориентированным на результат), который сокращает количество локальных переменных, а следовательно, упрощает логику:

```
var getValue = function getValue(object, propertyName) {
  return typeof object === 'undefined' ? undefined : object[propertyName]
}

var getNestedValueDeclaratively = function getNestedValueDeclaratively(object,
➥ propertyName) {
  return propertyName.split('.').reduce(getValue, object)
}
console.log(getNestedValueDeclaratively({bar: 'baz'}, 'bar') === 'baz')
console.log(getNestedValueDeclaratively({bar: { baz: 1 }}, 'bar.baz')=== 1)
```

Многие программисты учились писать код в императивном стиле, но обычно декларативный код проще. В данном случае сокращение числа переменных и команд помогает понять декларативный код с первого взгляда.

Впрочем, это был код JavaScript. А как насчет React? React применяет тот же декларативный подход при построении пользовательских интерфейсов. Сначала разработчик описывает элементы пользовательского интерфейса в декларативном стиле. А затем, когда в представлениях, генерируемых этими элементами, происходят изменения, React позаботится об обновлении. Вот так!

Удобство декларативного стиля React в полной мере проявляется тогда, когда вам приходится вносить изменения в представление. Они называются изменениями *внутреннего состояния*. При изменении состояния React соответствующим образом обновляет представление.

ПРИМЕЧАНИЕ О том, как работают состояния, рассказано в главе 4.

Во внутренней реализации React использует *виртуальную модель DOM* для определения различий (дельты) между текущим содержимым браузера и новым представлением. Этот процесс называется *поиском различий в DOM*, или *согласованием состояния с представлением* (при котором они перестают различаться). Это означает, что разработчикам не нужно беспокоиться об явном изменении представления; им достаточно обновить состояние, а представление будет обновляться автоматически по мере надобности.

И наоборот, с jQuery обновления приходится реализовывать в императивном стиле. Манипулируя с DOM, разработчик может на программном уровне изменять веб-страницу или ее отдельные части (более вероятный сценарий) без повторной перерисовки всей страницы. Собственно, при вызове методов jQuery происходят именно операции с DOM.

Некоторые фреймворки, например Angular, умеют автоматически обновлять представления. В Angular это называется *двусторонним связыванием данных* (two-way data binding), что фактически означает, что модели и представления связаны двусторонней передачей/синхронизацией данных.

Подходы jQuery и Angular оставляют желать лучшего по двум причинам. Их можно считать своего рода крайними случаями. В одном крайнем случае библиотека (jQuery) не делает ничего, а все обновления приходится вручную выполнять разработчику (то есть вам!). В другом крайнем случае фреймворк (Angular) делает абсолютно все.

Подход jQuery повышает риск ошибок и увеличивает объем работы. Кроме того, прямые манипуляции с обычной моделью DOM хорошо работают с простыми интерфейсами, но создают ограничения при большом количестве элементов в DOM-дереве. Это связано с тем, что результаты императивных функций труднее увидеть, чем результаты декларативных команд.

Подход Angular обсуждать сложнее, потому что с двусторонним связыванием ситуация быстро выходит из-под контроля. Вы вставляете все больше и больше логики, и внезапно разные представления начинают обновлять модели, а эти модели обновляют другие представления.

Да, код Angular читается несколько лучше, чем императивный стиль jQuery (и требует меньше ручного кодирования!), но здесь возникает другая проблема: Angular зависит от шаблонов и специального языка, использующего `ng`-директивы (например, `ng-if`). Недостатки Angular рассматриваются в следующем разделе.

Компонентная архитектура с использованием чистого JavaScript

Компонентная архитектура[1] (CBA) существовала еще до появления React. Разделение обязанностей, слабые связи и повторное использование кода лежат в основе этого подхода, обладающего многими преимуществами: программисты, в том числе и веб-разработчики, обожают компонентную архитектуру. Структурным элементом CBA в React является класс компонента. Как и другие CBA, он обладает многими преимуществами, главное из которых — повторное использование (ведь вы пишете меньше кода!).

Главное, чего не хватало до появления React, — реализации этой архитектуры на чистом JavaScript. Работая с Angular, Backbone, Ember и большинством других

[1] http://mng.bz/a65r.

MVC-подобных фреймворков клиентской части, вы используете один файл для JavaScript, а другой файл для шаблона (Angular использует термин *директивы* для компонентов). Использование двух языков (и двух и более файлов) для одного компонента создает ряд проблем.

Разделение HTML и JavaScript хорошо работает, если разметка HTML рендерится на сервере, а JavaScript используется только для эффектов типа мигающего текста. Теперь одностраничные приложения (SPA, Single Page Applications) обрабатывают сложный пользовательский ввод и выполняют рендер в браузере. Это означает, что HTML и JavaScript жестко связываются в функциональном отношении. Для разработчиков было бы удобнее, если бы им не приходилось разделять HTML и JavaScript в ходе работы над частью проекта (компонентом).

Возьмем код Angular, который отображает разные ссылки в зависимости от значения `userSession`:

```
<a ng-if="user.session" href="/logout">Logout</a>
<a ng-if="!user.session" href="/login">Login</a>
```

Код можно прочитать, но у читателя могут возникнуть сомнения относительно того, что получает `ng-if`: логическое значение или строку? Будет ли элемент скрыт или он вообще не рендерится? В случае Angular вы не можете быть уверены в том, будет ли элемент скрываться по истинному или ложному значению, если вы не знаете, как работает конкретная директива `ng-if`.

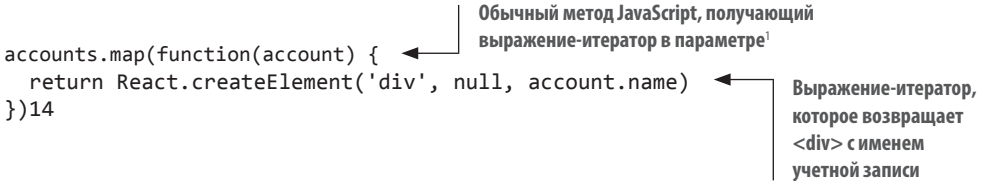
Сравните предыдущий фрагмент со следующим кодом React, который использует команду JavaScript `if/else` для условного рендеринга. Совершенно ясно, каким должно быть значение `user.session` и какой элемент (`logout` или `login`) будет рендериться при истинном значении. Почему? Потому что это обычный код JavaScript:

```
if (user.session) return React.createElement('a', {href: '/logout'}, 'Logout')
else return React.createElement('a', {href: '/login'}, 'Login')
```

Шаблоны полезны тогда, когда вам нужно перебрать массив данных и вывести свойство. Мы постоянно работаем со списками данных! Рассмотрим цикл `for` в Angular. Как упоминалось ранее, в Angular необходимо использовать специальный язык с директивами. Для цикла `for` используется директива `ng-repeat`:

```
<div ng-repeat="account in accounts">
  {{account.name}}
</div>
```

Один из недостатков шаблонов заключается в том, что разработчикам приходится изучать еще один язык. В React можно использовать чистый код JavaScript, а значит, вам не придется учить новый язык! Пример формирования пользовательского интерфейса для списка имен учетных записей на чистом JavaScript:



Представьте, что вы вносите изменения в список учетных записей. В списке нужно вывести номер учетной записи и другие поля. Как узнать, какими полями обладает учетная запись, кроме *имени*?

Придется открыть соответствующий файл JavaScript, который вызывает и использует этот шаблон, а затем найти записи *аккаунтов* для просмотра их свойств. Итак, вторая проблема с шаблонами: логика данных отделена от описания того, как эти данные должны рендериться.

Было бы намного лучше держать JavaScript и разметку в одном месте, чтобы вам не приходилось переключаться между файлами и языками. Именно так работает React; вскоре мы рассмотрим рендер элементов в React в примере Hello World.

> **ПРИМЕЧАНИЕ** Разделение обязанностей обычно относится к хорошим паттернам. В двух словах это означает разделение разных функций (работа с данными, уровень представления и т. д.). Работая с разметкой шаблонов и соответствующим кодом JavaScript, вы работаете над *одной функциональностью*. Вот почему наличие двух файлов (.js и html) не является разделением обязанностей.

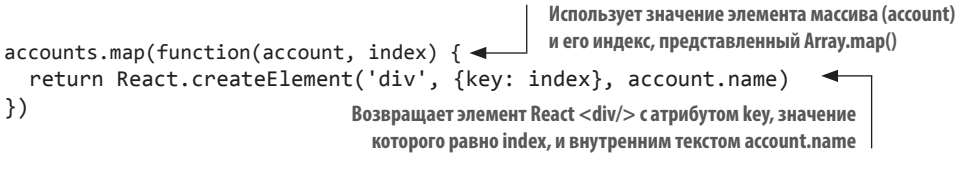
Если вы хотите явно задать метод для отслеживания элементов (например, для проверки того, что среди них нет дубликатов) в построенном списке, можно воспользоваться синтаксисом Angular `track by`:

```
<div ng-repeat="account in accounts track by account._id">
  {{account.name}}
</div>
```

Чтобы отслеживание осуществлялось по индексу массива, используйте `$index`:

```
<div ng-repeat="account in accounts track by $index">
{{account.name}}
</div>
```

Но мне и многим другим разработчикам не дает покоя вопрос: что это за волшебная переменная `$index`? В React в качестве значения атрибута `key` используется аргумент `map()`:



[1] http://mng.bz/555J.

Стоит заметить, что метод `map()` не является исключительной особенностью React. Вы можете использовать его с другими фреймворками, потому что он является частью языка. Однако благодаря декларативной природе `map()` идеально сочетается с React.

Я не придираюсь к Angular — это замечательный фреймворк. Дело в том, что если фреймворк использует предметно-ориентированный язык, вам придется изучать его специальные переменные и методы. В React можно использовать чистый JavaScript.

Если вы используете React, вы можете перенести свои знания в следующий проект даже в том случае, если в нем React не используется. С другой стороны, при использовании шаблонизатора X (или фреймворка Y со встроенным ядром шаблонов, использующим предметно-ориентированный язык), вы запираетесь в рамках этой системы и вам приходится называть себя разработчиком X/Y. Ваши знания не удастся перенести в проекты, не использующие X/Y. Подведем итог: компонентная архитектура на чистом JavaScript направлена на использование изолированных, хорошо инкапсулированных, пригодных для повторного использования компонентов, обеспечивающих улучшенное разделение обязанностей на основе функциональности, без необходимости в дополнительных предметно-ориентированных языках, шаблонах или директивах.

Работая со многими группами разработчиков, я заметил другой фактор, относящийся к простоте. React обладает более плавной, более постепенной кривой обучения по сравнению с фреймворками MVC (хотя React не базируется на MVC, поэтому я не буду их сравнивать) и шаблонизаторами, имеющими специальный синтаксис, например директивами Angular или Jade/Pug. Дело в том, что вместо использования мощи JavaScript большинство шаблонизаторов строит абстракции на базе собственного предметно-ориентированного языка, в каком-то смысле заново изобретая такие конструкции, как условия `if` или циклы `for`.

Мощные абстракции

В React используется мощная абстракция модели документа. Иначе говоря, React скрывает нижележащие интерфейсы и предоставляет нормализованные/синтезированные методы и свойства.

Например, при создании события `onClick` в React обработчик события получает не платформенный объект события для конкретного браузера, а синтетический объект события, который является оберткой для платформенных объектов событий. Вы можете рассчитывать на одинаковое поведение синтетических событий независимо от того, в каком браузере будет выполняться код. React также содержит набор синтетических событий для событий касания, которые отлично подходят для построения веб-приложений для мобильных устройств.

Другой пример абстракции DOM в React — возможность рендера элементов React на сервере. Данная возможность может пригодиться для улучшения поисковой оптимизации (SEO, Search Engine Optimization) и/или улучшения быстродействия.

При рендере компонентов React на сервере возможные варианты не ограничиваются DOM или строками HTML. Эти возможности будут рассмотрены в разделе 1.5.1. И если говорить о DOM, одним из самых заманчивых преимуществ React является превосходное быстродействие.

1.3.2. Скорость и удобство тестирования

Помимо необходимых обновлений DOM, ваш фреймворк может выполнить ненужные обновления, которые только снижают быстродействие сложных пользовательских интерфейсов. Эффект особенно заметен и неприятен для пользователя при большом количестве динамических UI-элементов на веб-странице.

С другой стороны, виртуальная модель DOM React существует только в памяти JavaScript. Каждый раз, когда происходит изменение в данных, React сначала сравнивает различия по своей виртуальной модели DOM; только когда библиотека знает, что в рендере произошли изменения, она обновляет фактическую модель DOM. На рис. 1.1 изображена высокоуровневая схема работы виртуальной модели DOM React при изменении данных.

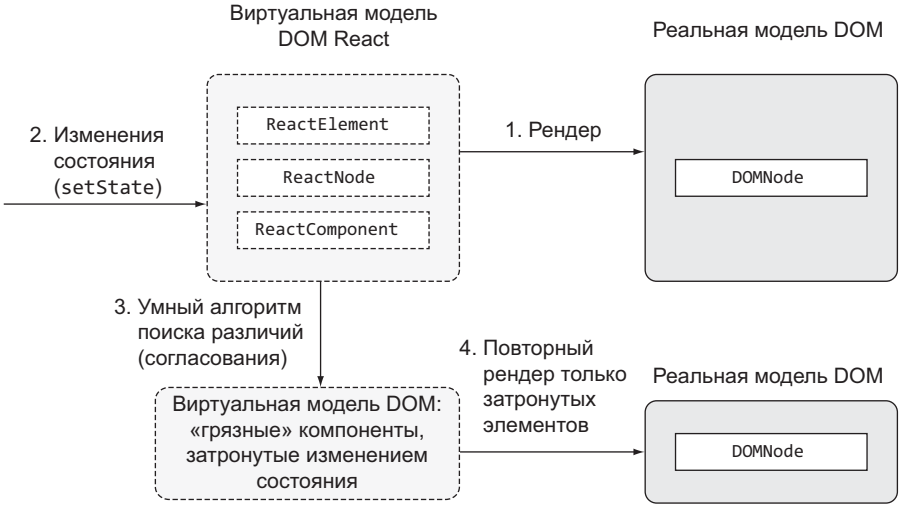


Рис. 1.1. После того как рендеринг компонента будет выполнен, при изменении состояния он сравнивается с виртуальной моделью DOM, находящейся в памяти, и в случае необходимости рендерится заново

В конечном итоге React обновляет только те части, для которых это абсолютно необходимо, чтобы внутреннее состояние (виртуальная модель DOM) и представление (реальная модель DOM) не отличались. Например, если присутствует элемент `<p>` и текст будет расширен состоянием компонента, обновлен будет только текст (то есть `innerHTML`), а не сам элемент. Таким образом достигается повышение

быстродействия по сравнению с повторным рендером целых наборов элементов или даже целых страниц (рендеринг на стороне сервера).

> **ПРИМЕЧАНИЕ** Если вас интересует техническая сторона алгоритмов и O большие, следующие две статьи отлично объясняют, как команде React удалось превратить задачу O(n3) в задачу O(n): «Reconciliation» на сайте React (http://mng.bz/PQ9X) и «React's Diff Algorithm» Кристофера Шедо (Christopher Chedeau) (http://mng.bz/68L4).

Еще одно преимущество виртуальной модели DOM — возможность проведения модульного тестирования без браузеров, не имеющих графического интерфейса, например PhantomJS (http://phantomjs.org). Существует Jest (https://facebook.github.io/jest), основанный на Jasmine (http://jasmine.github.io), который позволяет тестировать компоненты React прямо из командной строки!

1.3.3. Экосистема и сообщество

И наконец (но не в последнюю очередь), React поддерживается разработчиками исполинского веб-приложения Facebook, а также их коллегами из Instagram. Как и в случае с Angular и некоторыми другими библиотеками, поддержка технологии крупной компанией закладывает прочную основу для тестирования (так как оно открывается в миллионах браузеров), создает уверенность в будущем и повышает скорость реализации творческого вклада.

У React совершенно потрясающее сообщество. В большинстве случаев разработчикам даже не приходится самостоятельно реализовывать значительную часть кода. Обратите внимание на следующие ресурсы сообщества:

- Список компонентов React: https://github.com/brillout/awesome-react-components и http://devarchy.com/react-components.
- Набор компонентов React, реализующих спецификацию Google Material Design (https://design.google.com): http://react-toolbox.com.
- Компоненты React для Material Design: www.material-ui.com.
- Подборка компонентов React для взаимодействий в стиле Office и Office 360 (http://dev.office.com/fabric#/components), использующих язык Office Design Language: https://github.com/OfficeDev/office-ui-fabric-react.
- Подборка пакетов JS (в основном React) с открытым кодом: https://js.coach.
- Каталог компонентов React: https://react.rocks.
- Компоненты React из Khan Academy: https://khan.github.io/react-components.
- Каталог компонентов React: www.reactjsx.com.

Мой личный практический опыт с открытым кодом научил меня, что маркетинг проектов с открытым кодом не менее важен, ввиду широкого распространения

и успеха, чем сам код. Я имею в виду, что если у проекта убогий веб-сайт, не хватает документации и примеров, а логотип выглядит уродливо, большинство разработчиков не воспримет его серьезно, особенно в наши дни, когда библиотек JavaScript развелось так много. Разработчики привередливы, и они не станут выбирать библиотеку — «гадкого утенка».

Мой учитель любил говорить: «Не судите книгу по обложке». На первый взгляд это вроде бы противоречит сказанному выше. Но к сожалению, большинство людей, включая разработчиков программного обеспечения, склонны к предубеждениям, особенно когда это касается фирменного стиля. К счастью, React обладает отменной технической репутацией в отношении поддержки. И раз уж речь зашла о книгах и обложках — надеюсь, вы купили эту книгу не из-за обложки!

1.4. Недостатки React

Конечно, почти у всего есть недостатки. Это относится и к React, но полный список недостатков зависит от того, кого вы спросите. Некоторые различия, например декларативный стиль против императивного, в высшей степени субъективны. Таким образом, они могут считаться как достоинствами, так и недостатками. Ниже приведен мой список недостатков React (как и любые списки такого рода, он может быть необъективным, потому что он основан на мнениях, которые я слышал от других разработчиков):

- React не является полноценным фреймворком на все случаи жизни (вроде швейцарского ножа). Разработчику приходится использовать React в сочетании с такой библиотекой, как Redux или React Router, чтобы достичь функциональности, сравнимой с Angular или Ember. Это также может стать преимуществом, если вам нужна минималистская UI-библиотека для интеграции с существующим стеком.
- По зрелости React несколько уступает другим фреймворкам. Базовый API React продолжает изменяться, хотя после версии 0.14 изменений было очень мало; практические приемы работы с React (а также экосистема компонентов, плагины и дополнения) продолжают развиваться.
- React использует несколько нетрадиционный подход к веб-разработке, а JSX и Flux (часто используемые с React в качестве библиотеки данных) могут показаться устрашающими для новичков. Есть нехватка практических методов, хороших книг, учебных курсов и ресурсов для освоения React.
- React создает только одностороннее связывание. Хотя одностороннее связывание лучше для сложных приложений, поскольку оно устраняет большую часть сложностей, некоторые разработчики (особенно Angular-разработчики), привыкшие к двустороннему связыванию, увидят, что им приходится писать чуть больше кода. Я объясню, как работает одностороннее связывание по сравне-

нию с двусторонним связыванием Angular в главе 14, при рассмотрении работы с данными.

- React не является реактивным (в смысле реактивного программирования и архитектуры, которые в большей степени управляются событиями, обладают гибкостью и быстротой реакции) в исходном состоянии. Разработчику приходится использовать другие инструменты, такие как Reactive Extensions (RxJS, https://github.com/Reactive-Extensions/RxJS), для формирования асинхронных потоков данных с `Observable`.

Продолжая введение в React, посмотрим, как React интегрируется в веб-приложение.

1.5. Как React интегрируется в веб-приложение

Библиотеку React саму по себе, без React Router или библиотеки данных, в определенном смысле труднее сравнивать с фреймворками (такими, как Backbone, Ember и Angular) и лучше сравнивать с библиотеками для работы с пользовательскими интерфейсами, такими как шаблонизаторы (Handlebars, Blaze) и библиотеки для манипуляций с DOM (jQuery, Zepto). Собственно, многие команды заменили традиционные шаблонизаторы (такие, как Underscore в Backbone или Blaze в Meteor) на React, и притом с большим успехом. Например, команда PayPal перешла с Dust на Angular, как и многие другие компании, упоминавшиеся ранее в этой главе.

Вы можете использовать React только для части своего пользовательского интерфейса. Допустим, у вас имеется форма загрузки приложения на веб-странице, построенной на базе jQuery. Вы можете постепенно переводить это приложение на React, сначала преобразовав поля города и штата для автоматического заполнения на основании почтового индекса. Остальная часть формы может по-прежнему использовать jQuery. Если затем вы захотите продолжить, то можете перевести остальные поля формы с jQuery на React, пока вся страница не будет построена на базе React. Применяя аналогичный подход, многие команды успешно интегрировали React с Backbone, Angular или другим существующим фреймворком клиентской части.

React, по сути, независим от серверной части (back-end) для разработки клиентской части. Другими словами, вам не придется зависеть от Node.js или MERN (MongoDB, Express.js, React.js и Node.js) для использования React. Ничто не мешает использовать React с другими технологиями серверной части, такими как Java, Ruby, Go или Python. В конце концов, React — UI-библиотека. Ее можно интегрировать с любой серверной частью и любой библиотекой данных клиентской части (Backbone, Angular, Meteor и т. д.).

Подведем итог того, как React интегрируется в веб-приложения. React чаще всего используется в следующих ситуациях:

- Как UI-библиотека в одностраничных приложениях на базе стека, связанного с React, например React+React или Router+Redux.
- Как UI-библиотека (*V* в аббревиатуре MVC) в одностраничных приложениях на базе стека, не полностью связанного с React, например React+Backbone.
- Как подключаемый UI-компонент в любом стеке клиентской части (например, как компонент ввода с автозаполнением в стеке jQuery + стек рендера на стороне сервера).
- Как библиотека серверных шаблонов в полностью традиционном веб-приложении («толстый сервер») или в гибридном или изоморфно/универсальном веб-приложении, например сервере Express, использующем `express-react-views`.
- Как UI-библиотека в мобильных приложениях (например, iOS-приложение React Native).
- Как библиотека описания пользовательского интерфейса с другими объектами рендеринга (см. следующий раздел).

React хорошо работает с другими технологиями клиентской части, но в основном используется как часть одностраничной архитектуры (SPA), потому что одностраничная архитектура постепенно признается самым выигрышным или популярным решением для построения веб-приложением. О том, какое место React занимает в SPA, я расскажу в разделе 1.5.2.

В некоторых особых случаях React даже можно использовать *только на сервере* как своего рода шаблонизатор. Например, существует библиотека `express-react-views` (https://github.com/reactjs/express-react-views). Она рендерит представление на сервере из компонентов React. Рендер на стороне сервера возможен, потому что React позволяет использовать разные объекты рендеринга.

1.5.1. Библиотеки React и объекты рендеринга

В версии 0.14 и выше команда React разбивает библиотеку на два пакета: React Core (пакет `react` в `npm`) и ReactDOM (пакет `react-dom` в `npm`). Тем самым специалисты по сопровождению React ясно показали, что React находится на пути от простой библиотеки для веб-разработки к универсальной (иногда называемой изоморфной, потому что она может использоваться в разных средах) для описания пользовательских интерфейсов.

Например, в версии 0.13 в React существовал метод `React.render()` для монтирования элемента в узле DOM веб-страницы. В версиях 0.14 и выше необходимо включить `react-dom` и вызвать `ReactDOM.render()` вместо `React.render()`.

С многочисленными пакетами, созданными сообществом для поддержки разных объектов рендеринга, подход с отделением написания компонентов от рендеринга выглядит логично. Некоторые из этих модулей перечислены ниже:

- Визуализатор для интерфейса терминала blessed (https://github.com/chjj/blessed): http://github.com/Yomguithereal/react-blessed.
- Визуализатор для библиотеки ART (https://github.com/sebmarkbage/art): https://github.com/reactjs/react-art.
- Визуализатор для `<canvas>`: https://github.com/Flipboard/react-canvas.
- Визуализатор для 3D-библиотеки с использованием `three.js` (http://threejs.org): https://github.com/Izzimach/react-three.
- Визуализатор для виртуальной реальности и интерактивных панорамных сред: https://facebook.github.io/react-vr.

Кроме поддержки этих библиотек, отделение React Core от React-DOM упрощает совместное использование кода библиотеками React и React Native (используется для платформенной разработки мобильных iOS- и Android-приложений). В сущности, при использовании React для веб-разработки необходимо включить как минимум React Core и ReactDOM.

Кроме того, существуют дополнительные вспомогательные библиотеки React в React и `npm`. (До выхода React версии 0.15 некоторые из них входили в React как *дополнения*[1].) Эти вспомогательные библиотеки позволяют расширять функциональность, работать с неизменяемыми данными (https://github.com/kolodny/immutability-helper) и проводить тестирование.

Наконец, React почти всегда используется с JSX — крошечным языком, который позволяет разработчикам писать пользовательский интерфейс React в более выразительной форме. JSX можно транспилировать в обычный JavaScript с использованием Babel или другого аналогичного инструмента.

Как видите, архитектура имеет высокий уровень модульности — функциональность того, что относится к React, разбита на разные пакеты. Такой подход расширяет ваши возможности и предоставляет больше выбора, что хорошо. Никакая монолитная или расхваленная библиотека не приказывает действовать единственно возможным способом для реализации каких-либо вещей (подробнее об этом в разделе 1.5.3).

Если вы веб-разработчик, читающий эту книгу, вероятно, вы используете архитектуру SPA, либо вы уже построили веб-приложение на базе этой архитектуры и теперь хотите переработать его с React, либо вы пишете новый проект с нуля. А теперь рассмотрим место React в SPA как самого популярного подхода к построению веб-приложений.

---

[1] См. журнал изменений версии 15.5 со списком дополнений и библиотек npm: https://facebook.github.io/react/blog/2017/04/07/react-v15.5.0.html. Также информация о дополнениях доступна на следующей странице: https://facebook.github.io/react/docs/addons.html.

1.5.2. Одностраничные приложения и React

Другое название для архитектуры SPA — «толстый клиент»; браузер, выступая в роли клиента, содержит больше логики и выполняет такие функции, как рендеринг HTML, проверка данных, изменения пользовательского интерфейса и т. д. На рис. 1.2 показано общее представление типичной архитектуры SPA с пользователем, браузером и сервером. На иллюстрации изображен пользователь, который делает запрос, а также такие действия, как щелчки на кнопках, перетаскивание, наведение указателя мыши и т. д.:

1. Пользователь вводит URL-адрес в браузере, чтобы открыть новую страницу.
2. Браузер отправляет URL-запрос серверу.
3. Сервер отвечает статическими ассетами — HTML, CSS и JavaScript. В большинстве случаев разметка HTML минимальна, то есть это всего лишь заготовка веб-страницы. Обычно в это время на экране появляется сообщение «Загрузка...» и/или GIF-изображение вращающегося кольца.
4. Статические ассеты включают код JavaScript для SPA. При загрузке этот код выдает дополнительные запросы данных (запросы AJAX/XHR).
5. Данные возвращаются в JSON, XML или любом другом формате.
6. После того как одностраничное приложение получит данные, оно может заняться рендерингом отсутствующей разметки HTML (блок «Пользовательский интерфейс» на рисунке). Другими словами, рендер UI происходит в браузере средствами SPA-приложения, наполняющего шаблоны данными[1].

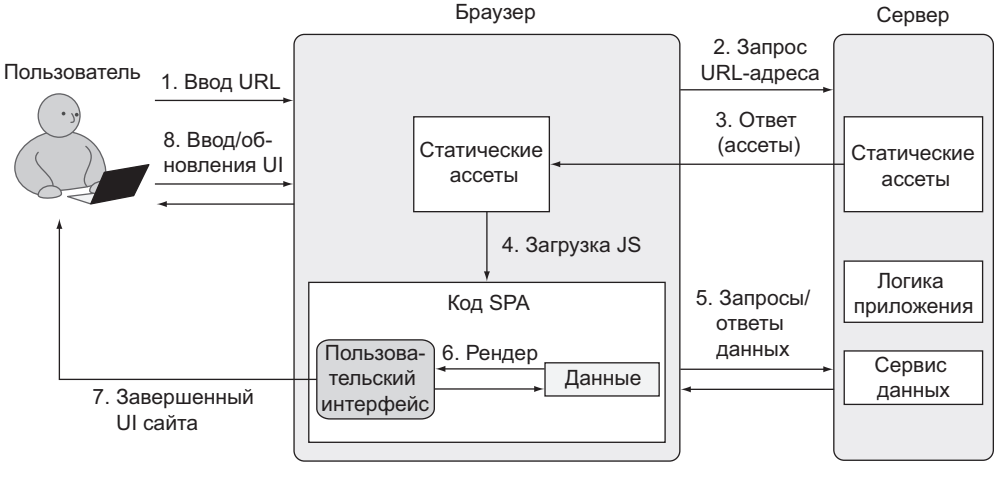


Рис. 1.2. Типичная архитектура SPA

[1] «What does it mean to hydrate an object?» — Stack Overflow, http://mng.bz/uP25.

7. После того как рендер в браузере завершится, SPA-приложение заменяет сообщение «Загрузка...», и пользователь может работать со страницей.
8. Пользователь видит красивую веб-страницу. Он может взаимодействовать со страницей («Ввод» на рисунке), инициировать новые запросы от SPA к серверу, и цикл шагов 2–6 продолжается. На этой стадии возможна маршрутизация в браузере, если SPA реализует ее; это означает, что навигация по новому URL-адресу инициирует не загрузку новой страницы с сервера, а повторный рендер SPA в браузере.

Подведем итог: в подходе SPA большая часть рендера UI происходит в браузере. От браузера и к нему передаются только данные. Сравните с подходом «толстого сервера», при котором весь рендер, или прорисовка, выполняется на сервере. (Здесь под «прорисовкой» имеется в виду генерирование разметки HTML шаблонами или кодом пользовательского интерфейса, а не простое отображение HTML в браузере.)

Обратите внимание: MVC-подобные архитектуры — подход самый популярный, но не единственный. React не требует использовать MVC-подобную архитектуру; но для простоты будем считать, что ваше SPA-приложение использует MVC-подобную архитектуру. Его возможные составляющие показаны на рис. 1.3. Навигатор (или библиотека маршрутизации) выполняет функции своеобразного контроллера в парадигме MVC; он указывает, какие данные следует запросить и какой шаблон должен использоваться. Навигатор/контроллер выдает запрос для получения данных, а затем заполняет шаблоны (представления) этими данными, чтобы визуализировать пользовательский интерфейс в форме HTML. Пользовательский интерфейс отправляет коду SPA действия: щелчки, наведение мыши, нажатия клавиш и т. д.

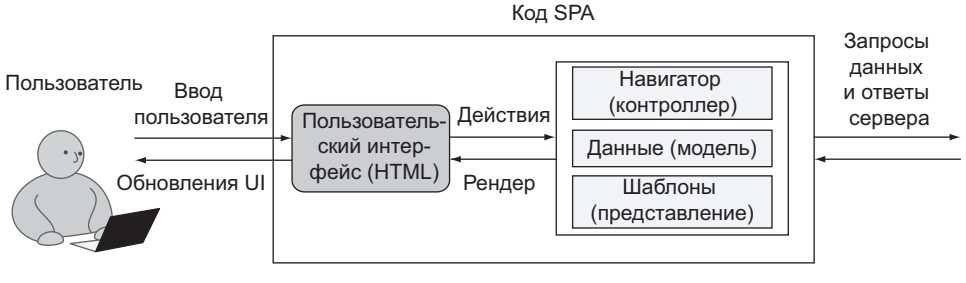


Рис. 1.3. Внутри одностраничного приложения

В архитектуре SPA данные интерпретируются и обрабатываются в браузере (рендер в браузере) и используются SPA для рендеринга дополнительной или изменения существующей разметки HTML. Так создаются удобные интерактивные веб-приложения, конкурирующие с настольными. Angular.js, Backbone.js и Ember.js — все это фреймворки клиентской части для построения SPA-приложений.

ПРИМЕЧНИЕ В разных фреймворках навигаторы, данные и шаблоны реализованы по-разному, поэтому рис. 1.3 применим не ко всем фреймворкам. Скорее, он демонстрирует самое распространенное разделение обязанностей в типичном SPA-приложении.

На диаграмме SPA на рис. 1.3 React занимает место в блоке «Шаблоны». React является уровнем представления, поэтому вы можете использовать его для генерирования HTML, предоставляя ему данные. Конечно, React делает намного больше, чем типичный шаблонизатор. Различия между React и другими шаблонизаторами (такими, как Underscore, Handlebars и Mustache) проявляются в способе разработки пользовательских интерфейсов, их обновления и управления их состоянием. Состояния будут более подробно рассмотрены в главе 4. А пока считайте состояния данными, которые могут изменяться и которые связаны с пользовательским интерфейсом.

1.5.3. Стек React

React не является полноценным фреймворком JavaScript для клиентской части. React выбирает минимализм и не заставляет вас каким-то конкретным образом выполнять такие операции, как моделирование данных, стилевое оформление или маршрутизация (то есть не относится к категории «безапелляционных»). По этой причине разработчикам приходится использовать React в паре с маршрутизацией и/или библиотекой моделирования.

Например, проект, в котором уже используется Backbone.js и шаблонизатор Underscore.js, может переключиться на Underscore для React, сохранив модели данных и маршрутизацию из Backbone. (Underscore также содержит вспомогательные инструменты, не только методы шаблонов. Вы можете использовать эти вспомогательные инструменты Underscore с React ради четкого декларативного стиля.)

В других случаях разработчики выбирают стек React, состоящий из данных и библиотек маршрутизации, созданных специально для использования с React:

- Библиотеки моделей данных и серверные части — RefluxJS (https://github.com/reflux/refluxjs), Redux (http://redux.js.org), Meteor (https://www.meteor.com) и Flux (https://github.com/facebook/flux).
- Библиотека маршрутизации — React Router (https://github.com/reactjs/react-router).
- Коллекция компонентов React для работы с библиотекой Twitter Bootstrap — React-Bootstrap (https://react-bootstrap.github.io).

Экосистема библиотек React растет с каждым днем. Кроме того, способность React описывать компоненты (автономные фрагменты пользовательского интерфейса) пригодится для повторного использования кода. Существует множество компонентов, упакованных как npm-модули. Просто для демонстрации того, что мелкие компоненты хорошо подходят для повторного использования, приведу список некоторых популярных компонентов React:

- Компонент для выбора даты: https://github.com/Hacker0x01/react-datepicker.
- Набор инструментов для рендера форм и проверки данных: https://github.com/prometheusresearch/react-forms.
- Компонент с автозаполнением, соответствующий стандарту WAI-ARIA: https://github.com/reactjs/react-autocomplete.

Затем есть JSX — пожалуй, один из самых частых аргументов против использования React. Если вы знакомы с Angular, то вам уже приходилось писать большой объем кода JavaScript в коде шаблона. Дело в том, что для современной веб-разработки простая разметка HTML слишком статична и сама по себе почти не используется. Мой совет: доверьтесь React и как следует опробуйте JSX.

JSX — маленький синтаксис для записи объектов React на объект с использованием скобок <>, как в XML/HTML. React хорошо сочетается с JSX, упрощая реализацию и чтение кода разработчиками. Считайте, что JSX — своего рода мини-язык, компилируемый в JavaScript. Итак, JSX не выполняется в браузере, а используется в качестве исходного кода для компиляции. Приведу компактный фрагмент, записанный на JSX:

```
if (user.session)
  return <a href="/logout">Logout</a>
else
  return <a href="/login">Login</a>
```

Даже если вы загрузите файл JSX в браузере с библиотекой, трансформирующей JSX в стандартный JavaScript в реальном времени, JSX при этом не выполняется; выполняться будет JavaScript. В этом смысле JSX напоминает CoffeeScript. Эти языки компилируются в стандартный JavaScript для использования расширенного синтаксиса и функциональности, которыми стандартный JavaScript не обладает.

Я знаю, что для некоторых читателей чередование XML с кодом JavaScript выглядит странно. Мне тоже пришлось какое-то время к этому привыкать, потому что я ожидал увидеть лавину сообщений о синтаксических ошибках. И еще — да, использовать JSX не обязательно. По этим двум причинам я не рассматриваю JSX до главы 3; но поверьте, когда вы освоите его, вы оцените, насколько он мощный.

А пока вы получили представление о том, что такое React, о стеке React и его месте в высокоуровневых SPA-приложениях. Пришло время взяться за дело и написать первый код React.

1.6. Первый код React: Hello world

Рассмотрим первый код React: типичнейший пример, встречающийся при изучении языков программирования, — приложение Hello world. (Если этого не сделать, боги программирования нас накажут!) Пока мы не будем использовать JSX, только

простой код JavaScript. Проект будет выводить заголовок «Hello world!!!» (`<h1>`) на веб-странице. На рис. 1.4 показано, как будет выглядеть проект после того, как вы его завершите (хотя, возможно, вы не испытываете такого энтузиазма и ограничитесь одним восклицательным знаком).

Рис. 1.4. Hello world

ИЗУЧЕНИЕ REACT БЕЗ JSX

Хотя большинство разработчиков React пишет на JSX, в браузерах выполняется только стандартный код JavaScript. Вот почему будет полезно понимать код React на стандартном JavaScript. Другая причина, по которой мы начинаем со стандартного JS, — чтобы показать, что JSX является необязательным, хотя, по сути, и стандартным языком React. Наконец, предварительная обработка JSX требует некоторого инструментария.

Я хочу, чтобы вы как можно быстрее начали работать с React, не тратя слишком много времени на подготовку в этой главе. Вся необходимая предварительная подготовка для JSX описана в главе 3.

Структура папок проекта проста. Она состоит из двух файлов JavaScript в папке `js` и одного файла HTML с именем `index.html`:

```
/hello-world
  /js
    react.js
    react-dom.js
  index.html
```

Два файла в папке `js` предназначены для библиотеки React версии 15.5.4[1]: `react-dom.js` (визуализатор DOM в браузере) и `react.js` (пакет React Core). Сначала необходимо

[1] Версия 15.5.4 — последняя на момент написания книги. Как правило, основные версии, такие как 14, 15 и 16, содержат значительные различия, тогда как дополнительные версии, такие как 15.5.3 и 15.5.4, содержат меньше критических изменений и конфликтов. Код книги был протестирован для версии 15.5.4. Код может работать с будущими версиями, но я не могу этого гарантировать, потому что никто не знает, что будет работать в будущих версиях, — этого не знают даже основные авторы.

загрузить упоминавшиеся ранее библиотеки React Core и ReactDOM. Это можно сделать несколькими способами. Я рекомендую воспользоваться файлами, входящими в архив исходного кода книги (www.manning.com/books/react-quickly и https://github.com/azat-co/react-quickly/tree/master/ch01/hello-world). Это самый надежный и простой способ, потому что он не зависит от других сервисов или инструментов. Другие способы загрузки React описаны в приложении А.

> **ПРЕДУПРЕЖДЕНИЕ** До выхода версии 0.14 эти две библиотеки включались в один пакет. Например, для версии 0.13.3 вам был нужен только файл react.js. В этой книге использовались React и React DOM версии 15.5.4 (последняя на момент написания книги), если только в тексте не указано обратное. Для большинства проектов части 1 понадобятся два файла: react.js и react-com.js. В главе 8 вам также понадобится библиотека prop-types (www.npmjs.com/package/prop-types), которая была частью React до версии 15.5.4, но теперь является отдельным модулем.

После того как файлы React будут размещены в папке js, создайте файл index.html в папке проекта hello-world. Файл HTML станет точкой входа приложения Hello World (то есть он будет открываться в браузере).

Разметка index.html очень проста. Она начинается с включения библиотек в `<head>`. В элементе `<body>` создается контейнер `<div>` с идентификатором `content` и элементом `<script>` (в котором позднее будет размещен код приложения), как показано в листинге 1.1.

Листинг 1.1. Загрузка библиотек и кода React (index.html)

```
<!DOCTYPE html>
<html>
  <head>
    <script src="js/react.js"></script> )
    <script src="js/react-dom.js"></script>
  </head>
  <body>
    <div id="content"></div>
    <script type="text/javascript">
      ...
    </script>
  </body>
</html>
```

Импортирует библиотеку React

Импортирует библиотеку ReactDOM

Определяет пустой элемент для подключения пользовательского интерфейса React

Начало кода React для приложения Hello World

Почему не выполнить рендер элемента React прямо в элементе `<body>`? Потому что это может создать конфликт с другими библиотеками и браузерными расширениями, которые манипулируют с телом документа. Более того, попытавшись присоединить элемент непосредственно к телу документа, вы получите предупреждение:

```
Rendering components directly into document.body is discouraged...
```

Это еще одна сильная сторона React: отличные предупреждения и сообщения об ошибках!

ПРИМЕЧАНИЕ Предупреждения и сообщения об ошибках React не включаются в итоговую сборку для сокращения информационного шума, повышения безопасности и сокращения размера дистрибутива. Итоговая сборка представляет собой минифицированный файл из библиотеки React Core, например react.min.js. Версия для разработки с предупреждениями и сообщениями об ошибках не минифицирована (например, react.js).

Включая библиотеки в файле HTML, вы получаете доступ к глобальным объектам React и ReactDOM: `window.React` и `window.ReactDOM`. Вам понадобятся два метода этих объектов: для создания элемента (React) и для рендера его в контейнере `<div>` (ReactDOM), как показано в листинге 1.2.

Чтобы создать элемент React, достаточно вызвать `React.createElement(elementName, data, child)` с тремя аргументами:

- `elementName` — HTML в виде строки (например, `'h1'`) или класс нестандартного компонента в виде объекта (например, HelloWorld; см. раздел 2.2).
- `data` — данные в форме атрибутов и свойств (свойства будут рассмотрены позднее), например `null` или `{name: 'Azat'}`.
- `child` — дочерний элемент или внутренний контент HTML/text, например Hello world!

Листинг 1.2. Создание и прорисовка элемента h1 (index.html)

```
var h1 = React.createElement('h1', null, 'Hello world!')
ReactDOM.render(
  h1,
  document.getElementById('content')
)
```

Создает и сохраняет в переменной элемент React типа h1

Прорисовывает элемент h1 в реальном элементе DOM с идентификатором «content»

В этом листинге программа получает элемент React типа `h1` и сохраняет ссылку на этот объект в переменной `h1`. Переменная `h1` не содержит реальный узел DOM, это воплощение компонента React `h1` (элемент). Вы можете присвоить ему любое имя на свое усмотрение, например `helloWorldHeading`. Иначе говоря, React предоставляет абстракцию над DOM.

ПРИМЕЧАНИЕ Имя переменной `h1` выбрано произвольно. Вы можете использовать любое имя по своему усмотрению (хоть `bananza`) при условии, что вы используете ту же переменную в `ReactDOM.render()`.

После того как элемент будет создан и сохранен в `h1`, он рендерится в узле/элементе DOM с идентификатором `content` при помощи метода `ReactDOM.render()`, показан-

ного в листинге 1.2. При желании переменную h1 можно переместить в вызов render. Результат будет тем же, но в этом случае лишняя переменная не используется:

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

Теперь откройте файл index.html, предоставляемый статическим веб-сервером HTTP, в своем любимом браузере. Я рекомендую использовать обновленную версию Chrome, Safari или Firefox. В браузере должна открыться страница с сообщением «Hello world!», как показано на рис. 1.5.

На рисунке показана вкладка Elements в Chrome DevTools с выбранным элементом <h1>. Обратите внимание на атрибут data-reactroot; он показывает, что этот элемент был сгенерирован ReactDOM.

Кстати говоря, код React (листинг 1.2) можно выделить в отдельный файл, вместо того чтобы создавать элементы и рендерить их методом ReactDOM.render() в файле index.html (листинг 1.1). Например, вы можете создать файл script.js и скопировать в него элемент h1 и вызов ReactDOM.render(). Затем файл script.js включается в index.html после контейнера с идентификатором content:

```
<div id="content"></div>
<script src="script.js"></script>
```

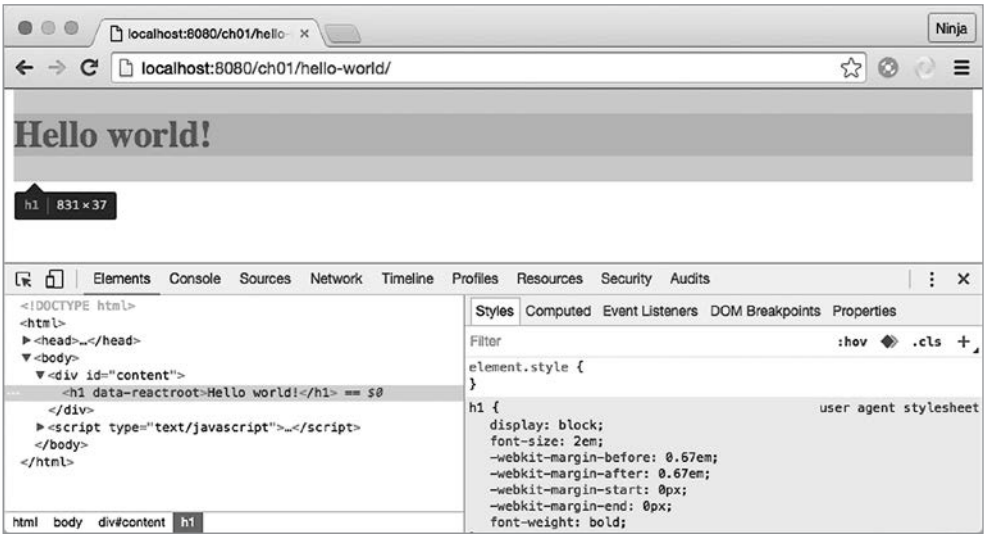


Рис. 1.5. Анализ приложения Hello world, сгенерированного React

ЛОКАЛЬНЫЙ ВЕБ-СЕРВЕР РАЗРАБОТКИ

Вместо того чтобы открывать файл index.html в браузере напрямую, лучше использовать локальный веб-сервер, потому что с веб-сервером ваши приложения JavaScript смогут выдавать AJAX/XHR запросы. Чтобы определить источник данных, достаточно посмотреть на URL в адресной строке. Если адрес начинается с префикса file, это файл; если адрес начинается с префикса http, это сервер. Эта функция понадобится вам для будущих проектов. Как правило, локальный веб-сервер HTTP прослушивает входящие запросы по адресу 127.0.0.1 или localhost.

Использовать можно любой веб-сервер с открытым кодом — Apache, MAMP (мои любимые, потому что они написаны на Node.js), node-static (https://github.com/cloudhead/node-static) или http-server (www.npmjs.com/package/http-server). Для установки node-static или http-server необходимо предварительно установить Node.js и npm. Если они еще не установлены, инструкции по установке для Node и npm можно найти в приложении А или по адресу http://nodejs.org.

Если же Node.js и npm уже установлены на вашей машине, выполните команду `npm i -g node-static` или `npm i -g http-server` в терминале или командной строке. Перейдите в папку с исходным кодом и запустите `static` или `http-server`. На своей машине я запускаю `static` из папки react-quickly, поэтому в адресной строке моего браузера следует ввести путь к Hello World: http://localhost:8080/ch01/hello-world/ (см. рис. 1.5).

Поздравляю! Вы только что успешно реализовали свой первый код React!

1.7. Вопросы

1. Декларативный стиль программирования не разрешает изменять хранимые значения. Он работает по принципу «вот что мне нужно», в отличие от принципа «вот как это следует делать». Да или нет?
2. Какой из следующих методов осуществляет рендеринг компонентов React в DOM (внимание, каверзный вопрос): `ReactDOM.renderComponent`, `React.render`, `ReactDOM.append` или `ReactDOM.render`?
3. Чтобы использовать React в одностраничных приложениях, необходимо задействовать на сервере Node.js. Да или нет?
4. Для рендера элементов React на веб-странице необходимо включить файл react-com.js. Да или нет?
5. React решает задачу обновления представлений в соответствии с изменениями данных. Да или нет?

1.8. Итоги

- React использует декларативный стиль; это всего лишь представление или слой пользовательского интерфейса.
- React использует компоненты, которые создаются вызовом `ReactDOM.render()`.
- Классы компонентов React создаются при помощи `class` и его обязательного метода `render()`.
- Компоненты React пригодны для повторного использования и получают неизменяемые свойства, для обращения к которым используется синтаксис `this.props.NAME`.
- Для разработки и составления пользовательских интерфейсов в React используется чистый JavaScript.
- Использовать JSX (и XML-подобный синтаксис объектов React) не обязательно; в разработке на базе React вы применяете JSX по своему желанию!
- Итог: React в веб-разработке состоит из библиотек React Core и ReactDOM. React Core — библиотека, ориентированная на построение и совместное использование компонентов пользовательского интерфейса с помощью JavaScript и (возможно) JSX в изоморфном/универсальном стиле. С другой стороны, для работы с React в браузере можно использовать библиотеку ReactDOM, которая содержит методы как для рендера DOM, так и для рендера на стороне сервера.

1.9. Ответы

1. Да. Декларативный стиль определяет «что мне нужно», а императивный — «как это следует делать».
2. `ReactDOM.render`.
3. Нет. Вы можете использовать любую технологию серверной части.
4. Да. Библиотека ReactDOM необходима.
5. Да. Это главная задача, которую решает React.

2 Первые шаги с React

Посмотрите вступительный видеоролик к этой главе, отсканировав QR-код или перейдя по ссылке http://reactquickly.co/videos/ch02.

ЭТА ГЛАВА ОХВАТЫВАЕТ СЛЕДУЮЩИЕ ТЕМЫ:

- Вложение элементов.
- Создание класса компонента.
- Работа со свойствами.

В этой главе вы сделаете свои первые шаги с React и заложите фундамент для всех последующих глав. Для разработчика очень важно понимать такие концепции React, как элементы и компоненты. По сути, *элементы* являются экземплярами *компонентов* (также называемых классами компонентов). Когда они используются и для чего они нужны? Скоро узнаете!

ПРИМЕЧАНИЕ Исходный код примеров этой главы доступен по адресам www.manning.com/books/react-quickly и https://github.com/azat-co/react-quickly/tree/master/ch02 (в папке ch02 репозитория GitHub https://github.com/azat-co/react-quickly). Также некоторые демонстрационные примеры доступны по адресу http://reactquickly.co/demos.

2.1. Вложение элементов

В предыдущей главе вы научились создавать элементы React. Напомним, что для этой цели используется метод `React.createElement()`. Например, элемент ссылки может создаваться так:

```
let linkReactElement = React.createElement('a',
  {href: 'http://webapplog.com'},
  'Webapplog.com'
)
```

Проблема в том, что большинство пользовательских интерфейсов обычно содержат несколько элементов (например, ссылка может находиться внутри меню). Так, на рис. 2.1 показаны кнопки, миниатюры видеороликов и проигрыватель YouTube.

Решением для создания более сложных структур с иерархической природой является вложение элементов. В предыдущей главе вы реализовали свой первый код React, создав элемент React `h1` и отрендерив его в DOM методом `ReactDOM.render()`:

```
let h1 = React.createElement('h1', null, 'Hello world!')
ReactDOM.render(
  h1,
  document.getElementById('content')
)
```

Важно заметить, что `ReactDOM.render()` получает в аргументе только один элемент — `h1` в данном примере (рис. 2.2).

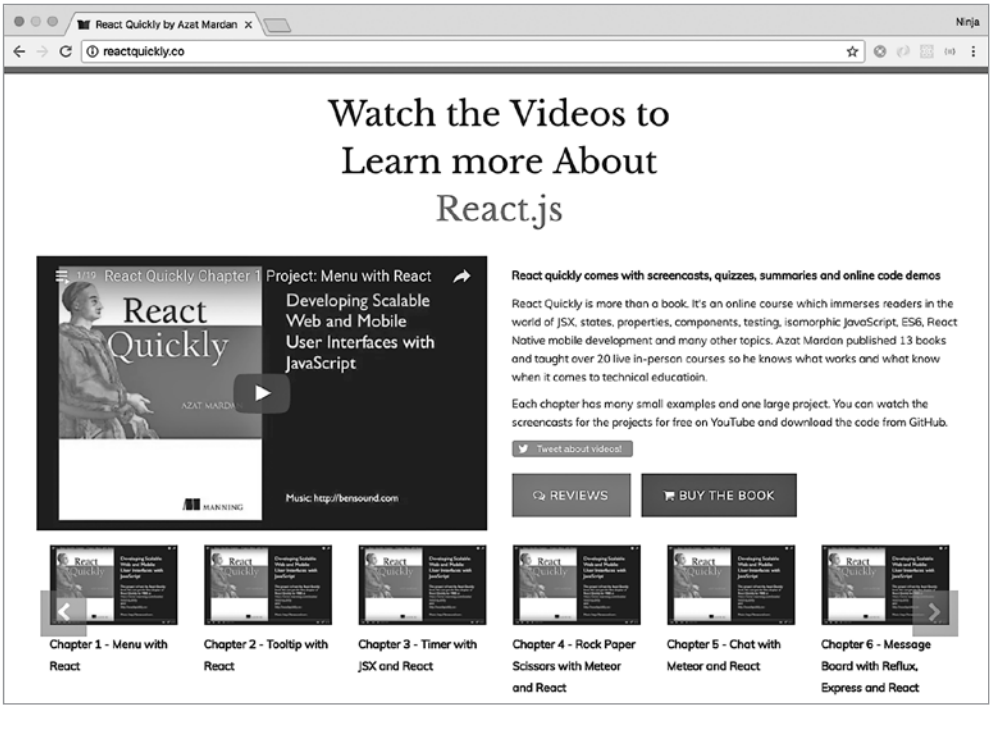


Рис. 2.1. Веб-сайт книги содержит много вложенных элементов пользовательского интерфейса

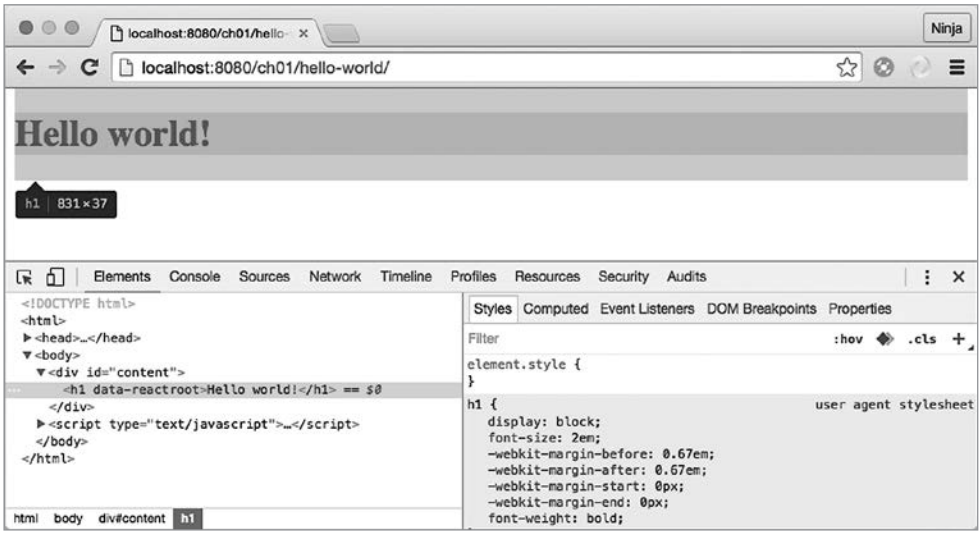


Рис. 2.2. Рендер элемента заголовка

Как упоминалось в начале раздела, проблема возникает, когда потребуется отрендерить два элемента одного уровня (например, два элемента `h1`). В этом случае элементы можно упаковать в визуально нейтральный элемент, как показано на рис. 2.3. Обычно для этого хорошо подходит контейнер `<div>` или `<span>`.

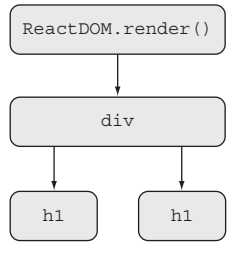


Рис. 2.3. Определение иерархической структуры при рендере React с использованием контейнера для одноуровневых заголовков

При вызове `createElement()` можно передавать неограниченное количество параметров. Все параметры, после второго, становятся дочерними элементами. Эти дочерние элементы (`h1` в данном случае) становятся *одноуровневыми*, то есть они находятся на одном уровне иерархии по отношению друг к другу, как можно увидеть на рис. 2.4, используя инструментарий DevTools в Chrome.

С учетом сказанного воспользуемся методом `createElement()` для создания элемента `<div>` с двумя дочерними элементами `<h1>` (ch02/hello-world-nested/index.html).

Листинг 2.1. Создание элемента с двумя дочерними элементами <h1>

```
let h1 = React.createElement('h1', null, 'Hello world!')
ReactDOM.render(
  React.createElement('div', null, h1, h1),
  document.getElementById('content')
)
```

Если в третьем параметре createElement() передается строка, то она задает текстовое значение создаваемого элемента

Если третий и последующие параметры содержат не текст, то эти параметры задают дочерние элементы для создаваемого элемента

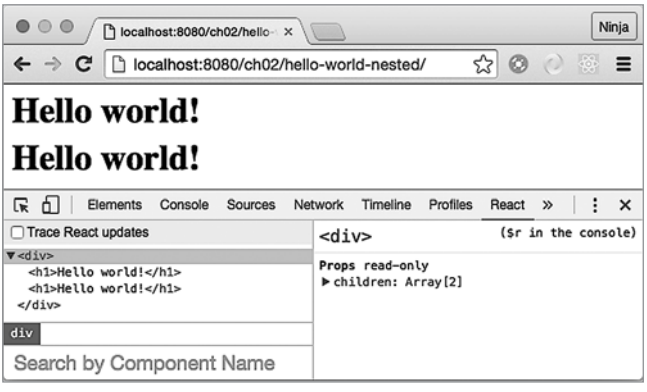


Рис. 2.4. Контейнер для вложенных одноуровневых элементов h1 в React DevTools

REACT DEVELOPER TOOLS

Кроме вкладки Elements, включенной в Chrome DevTools по умолчанию, вы можете установить расширение (или плагин) с именем React Developer Tools (последняя вкладка на рис. 2.4). React Developer Tools также существует для Firefox. Расширение позволяет детально анализировать результаты рендера React, включая иерархию компонентов, имена, свойства, состояния и многое другое.

Репозиторий GitHub находится по адресу https://github.com/facebook/react-devtools. Также React Developer Tools для Chrome можно найти по адресу http://mng.bz/V276, а для Firefox — по адресу http://mng.bz/59V9.

Разметка HTML остается такой же, как в примере Hello world из главы 1, — при условии, что вы включите необходимые библиотеки React и ReactDOM и в модели присутствует узел `content` (ch02/hello-world-nested/index.html).

Листинг 2.2. Разметка HTML для вложенных элементов без кода React

```
<!DOCTYPE html>
<html>
  <head>
    <script src="js/react.js"></script>
    <script src="js/react-dom.js"></script>
  </head>
  <body>
    <div id="content"></div>
    <script type="text/javascript">
    ...
    </script>
  </body>
</html>
```

До сих пор в первом параметре `createElement()` передавались только строковые значения. Но первый параметр может получать данные двух типов:

- Стандартные теги HTML в виде строки, например `'h1'`, `'div'` или `'p'` (без угловых скобок). Имя записывается в нижнем регистре.
- Классы компонентов React в виде объектов, например `HelloWorld`. Имя записывается с буквы верхнего регистра.

В первом случае генерируются стандартные элементы HTML. React перебирает свой список стандартных элементов HTML и при обнаружении совпадения использует его как тип элемента React. Например, при передаче строки `'p'` React найдет совпадение, потому что `p` — имя тега. В результате при прорисовке этого элемента React в DOM появится тег `<p>`.

Теперь рассмотрим второй тип входных данных: создание и передачу нестандартных классов компонентов.

2.2. Создание классов компонентов

После вложения элементов в React появляется следующая проблема: количество элементов стремительно растет. Приходится использовать компонентную архитектуру, описанную в главе 1, которая позволяет повторно использовать код за счет разбиения функциональности на слабо связанные составляющие. На помощь приходят *классы компонентов* (или просто *компоненты*, как они часто называются для краткости, — не путайте с веб-компонентами).

Стандартные теги HTML можно рассматривать как строительные блоки. Вы можете использовать их для построения ваших собственных классов компонентов React, которые затем используются для создания нестандартных элементов (экземпляров классов). Используя нестандартные элементы, можно инкапсулировать и абстрагировать логику в импортируемых классах (составных компонентах, пригодных для повторного использования). Эта абстракция позволяет командам повторно использовать пользовательские интерфейсы в больших сложных приложениях, а также в разных проектах. Примеры такого рода — компоненты с автозаполнением, инструментарии, меню и т. д.

Создание элемента ‘Hello world!’ с тегом HTML в методе `createElement()` выглядело достаточно просто:

```
(createElement('h1', null, 'Hello World!')
```

Но что, если вы хотите вынести Hello world в отдельный класс, как показано на рис. 2.5? Допустим, вы хотите заново использовать Hello world в десяти разных проектах! (Пожалуй, это перебор, но хороший компонент с автозаполнением определенно может использоваться повторно.)

Интересно, что класс компонента React создается расширением класса `React.Component` в синтаксисе E6 `class CHILD extends PARENT`. Создадим нестандартный класс компонента HelloWorld с использованием синтаксиса `class HelloWorld extends React.Component`.

Единственный аспект нового класса, который обязательно необходимо реализовать, — это метод `render()`. Этот метод *должен* возвращать один элемент React, `createElement()`, который создается из другого нестандартного класса компонента или тега HTML. Оба могут содержать вложенные элементы.

В листинге 2.3 (ch02/hello-world-class/js/script.js) показано, как переработать пример с вложением Hello World (листинг 2.1) в приложение с нестандартным классом компонента React, HelloWorld. Преимущество такого решения заключается в том, что нестандартный класс позволяет более эффективно использовать этот пользовательский интерфейс повторно. Обязательный метод `render()` компонента HelloWorld возвращает тот же элемент `<div>` из предыдущего примера. После того как вы создадите нестандартный класс `HelloWorld`, его можно будет передать в виде объекта (не в виде строки) `ReactDOM.render()`.

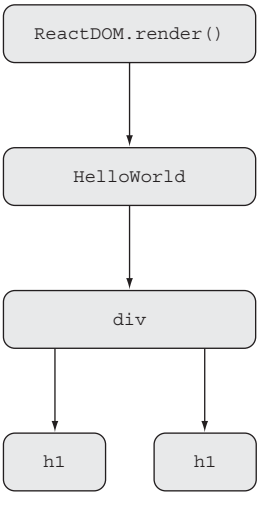


Рис. 2.5. Рендеринг элемента , созданного из нестандартного класса компонента, вместо его прямого рендера

Листинг 2.3. Создание и прорисовка класса компонента React

```
let h1 = React.createElement('h1', null, 'Hello world!')
class HelloWorld extends React.Component {
  render() {
    return React.createElement('div', null, h1, h1)
  }
}
ReactDOM.render(
  React.createElement(HelloWorld, null),
  document.getElementById('content')
)
```

Создает метод render() в виде выражения (функция, возвращающая один элемент)

Определяет класс компонента React с именем, начинающимся с символа верхнего регистра

Реализует команду return с одним элементом React, чтобы класс React мог вызвать render() и получить элемент с двумя элементами h1

Использует класс HelloWorld для создания элемента, при этом в первом аргументе передается объект вместо строки

Присоединяет элемент React к реальному элементу DOM с идентификатором «content»

Имена переменных, содержащих компоненты React, принято записывать с символа верхнего регистра. В стандартном JS это не обязательно (имя переменной можно начать с символа нижнего регистра `helloWorld`), но поскольку в JSX это обяза-

тельно, в данном случае это соглашение применяется. (В JSX React использует верхний и нижний регистр для того, чтобы отличать нестандартные компоненты вроде `<HelloWorld/>` от обычных элементов HTML `<h1/>`. Однако в стандартном JS это различие обеспечивается передачей переменной, такой как `HelloWorld`, либо строки, такой как `'h1'`. Чем раньше вы привыкнете к использованию соглашения о верхнем регистре для нестандартных компонентов, тем лучше.) JSX более подробно рассматривается в главе 3.

ES6+/ES2015+ И REACT

Пример с классом компонента определяет `render()` в стиле ES6, при котором опускается двоеточие и слово function. Это то же самое, как если бы вы определяли атрибут (также встречается термин «ключ» или «свойство объекта») со значением, которое представляет собой функцию, то есть `render: function()`. Лично я предпочитаю — да и вам рекомендую использовать — стиль методов ES6, потому что он короче (чем меньше символов вы вводите, тем меньше будет ошибок).

Исторически React использовал для создания классов компонентов собственный метод `React.createClass()`. Существуют небольшие различия между использованием класса ES6 для расширения `React.Component` и использования `React.createClass()`. Обычно вы используете либо class (рекомендуется), либо `createClass()`, но только что-то одно. Более того, в React 15.5.4 метод `createClass()` считается устаревшим (то есть более не поддерживается).

И хотя метод `React.createClass()` все еще используется некоторыми командами, в мире React проявляется общая тенденция к переходу на единый стандарт: использование варианта с классом ES6. Эта книга смотрит в будущее и использует самые популярные инструменты и решения, поэтому в ней основное внимание уделяется ES6. Примеры в стиле ES5 для некоторых проектов (они снабжены префиксом -es5) можно найти в репозитории GitHub; они предназначались для ранней версии этой книги.

По состоянию дел на август 2016 года почти все современные браузеры поддерживали эти (и почти все остальные) возможности ES6 по умолчанию (не требуя дополнительных инструментов[1]), поэтому я предполагаю, что вы с ними знакомы. Если нет (или вам потребуется освежить в памяти/получить дополнительную информацию о ES6+/ES2015+ и основных возможностях, относящихся к React), обращайтесь к приложению Д или специализированной литературе, например книге «Exploring ES6» Акселя Раушмайера (Axel Rauschmayer) (бесплатная электронная версия доступна по адресу http://exploringjs.com/es6).

По аналогии с методом `ReactDOM.render()`, метод `render()` из `createClass()` может *возвращать только один элемент*. Если вам нужно вернуть несколько элементов

[1] ECMAScript 6 Compatibility Table, https://kangax.github.io/compat-table/es6.

одного уровня, упакуйте их в контейнер `<div>` или другой нетребовательный элемент (например, `<span>`). Вы можете выполнить код в браузере; результат показан на рис. 2.6.

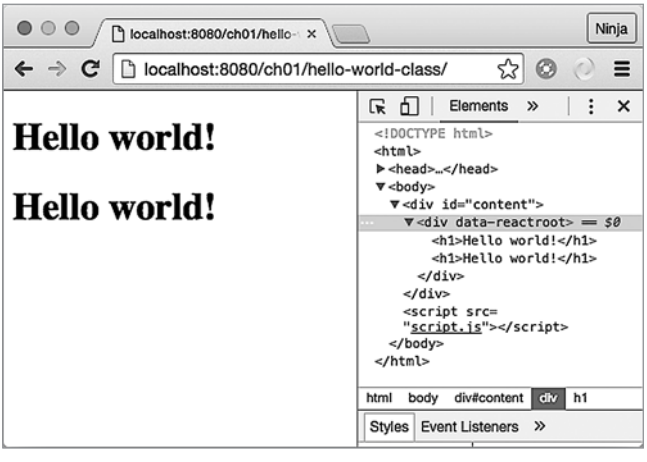


Рис. 2.6. Рендер элемента, созданного на базе нестандартного класса компонента HelloWorld

Казалось бы, переработка особых результатов не принесла; но что, если вам потребуется вывести больше сообщений Hello world? Для этого можно многократно воспользоваться компонентом `HelloWorld` и упаковать экземпляры в контейнер `<div>`:

```
...
ReactDOM.render(
  React.createElement(
    'div',
    null,
    React.createElement(HelloWorld),
    React.createElement(HelloWorld),
    React.createElement(HelloWorld)
  ),
  document.getElementById('content')
)
```

Такова сила повторного использования компонентов! Оно ускоряет разработку и сокращает количество ошибок. Компоненты также имеют события жизненного цикла, события DOM и другие возможности, благодаря которым они становятся интерактивными и автономными; эти темы рассматриваются в следующих главах.

А пока все элементы `HelloWorld` будут выглядеть одинаково. Можно ли как-то настроить их? Как бы задать атрибуты элементов и изменить их контент и/или поведение? На помощь приходят свойства.

2.3. Работа со свойствами

Свойства — краеугольный камень декларативного стиля, используемого в React. Свойства можно рассматривать как неизменяемые значения внутри элемента. Они позволяют модифицировать элементы, используемые в представлениях: например, вы можете изменить URL-адрес ссылки, передав новое значение свойства:

```
React.createElement('a', {href: 'http://node.university'})
```

Помните, что свойства *неизменяемы в пределах своих компонентов*. Родитель назначает свойства своих дочерних элементов при их создании. Дочерний элемент не должен изменять свои свойства. (*Дочерним* называется элемент, вложенный внутрь другого элемента; например, `<h1/>` является потомком `<HelloWorld/>`.) Например, свойство `PROPERTY_NAME` со значением `VALUE` может передаваться следующим образом:

```
<TAG_NAME PROPERTY_NAME=VALUE/>
```

Свойства сильно напоминают атрибуты HTML. Это одно из их применений, но есть и другое: свойства элемента можно использовать в коде так, как вы считаете нужным. Возможные варианты использования свойств:

- Для генерирования стандартных атрибутов HTML элемента: `href`, `title`, `style`, `class` и т. д.
- В коде JavaScript класса компонента React при использовании значений `this.props`, например `this.props.PROPERTY_NAME` (где `PROPERTY_NAME` — произвольное имя).

Во внутренней реализации React проверяет имя свойства (`PROPERTY_NAME`) по списку стандартных атрибутов. Если совпадение будет обнаружено, то свойство преобразуется в атрибут элемента (первый сценарий). Значение атрибута также доступно в форме `this.props.PROPERTY_NAME` в коде класса компонента.

Если совпадение среди стандартных имен атрибутов HTML не будет обнаружено (второй сценарий), значит, имя свойства не является стандартным атрибутом. Оно не будет преобразовано в атрибут элемента. Однако значение при этом все равно остается доступным в объекте `this.props` (например, `this.props.PROPERTY_NAME`). Вы можете использовать его в коде или выполнить его явную обработку в методе `render()`. При этом разным экземплярам одного класса могут передаваться разные данные. Это позволяет вам повторно использовать компоненты, потому что вы можете на программном уровне изменять рендер элементов, передавая разные свойства.

OBJECT.FREEZE() И OBJECT.ISFROZEN()

Во внутренней реализации React использует вызов `Object.freeze()`[1] из ES5 для того, чтобы сделать объект `this.props` неизменяемым. Чтобы проверить, был ли объект объявлен неизменяемым, можно воспользоваться методом `Object.isFrozen()`[2]. Например, вы можете проверить, вернет ли следующая команда true:

```
class HelloWorld extends React.Component {
  render() {
    console.log(Object.isFrozen(this.props))
    return React.createElement('div', null, h1, h1)
  }
}
```

Если вам понадобится более подробная информация, я рекомендую обратиться к журналу изменений React[3] и поискать информацию в GitHub-репозитории React[4].

С этой возможностью свойств можно пойти еще дальше и полностью изменять сгенерированные элементы в зависимости от значения свойства. Например, если свойство `this.props.heading` истинно, то текст «Hello» генерируется в виде заголовка, а если ложно — то как обычный абзац:

```
render() {
  if (this.props.heading) return <h1>Hello</h1>
  else return <p>Hello</p>
}
```

Иначе говоря, если вы используете один компонент с разными наборами свойств, элементы, сгенерированные для этого компонента, могут различаться. Свойства могут генерироваться в `render()`, использоваться в коде компонентов или в качестве атрибутов HTML.

Чтобы продемонстрировать использование свойств компонентов, мы слегка изменим Helloworld с `props`. Наша цель — повторно использовать компонент HelloWorld так, чтобы для каждого экземпляра класса генерировался разный текст и разные атрибуты HTML. Заголовки `HelloWorld` (тег `<h1>`) будут расширены тремя свойствами (рис. 2.7):

- `id` — соответствует стандартному атрибуту `id` и прорисовывается React автоматически.

[1] Mozilla Developer Network, Object.freeze(), http://mng.bz/p6Nr.
[2] Mozilla Developer Network, Object.isFrozen(), http://mng.bz/0P75.
[3] GitHub, 2016-04-07-react-v15, http://mng.bz/j6c3.
[4] GitHub, результаты поиска «freeze», http://mng.bz/2l0Z.

- `frameworkName` — не соответствует ни одному стандартному атрибуту `<h1>`, но выводится в тексте заголовков.
- `title` — соответствует стандартному атрибуту `title` и рендерится React автоматически.

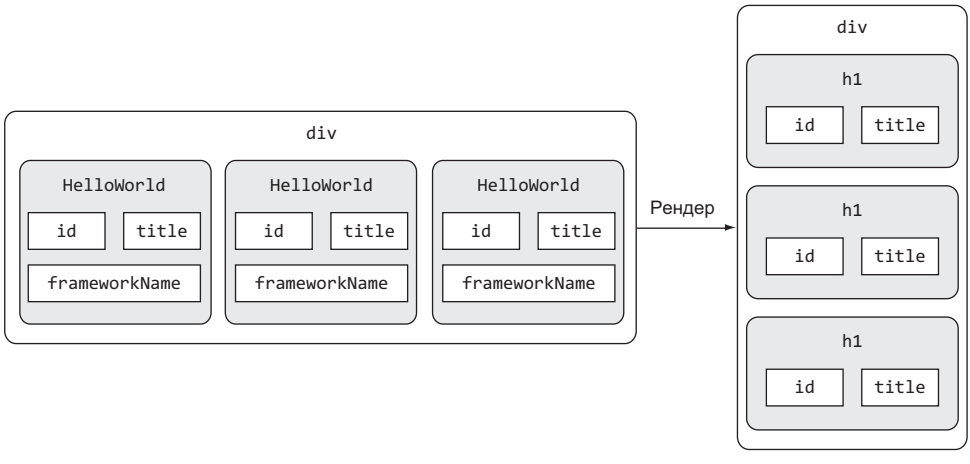


Рис. 2.7. Класс компонента HelloWorld рендерит свойства, представляющие стандартные атрибуты HTML, но не свойство frameworkName

Если имя свойства соответствует стандартному атрибуту HTML, то оно будет отрендерено как атрибут элемента `<h1>` (рис. 2.7). Таким образом, два свойства `id` и `title` будут отрендерены как атрибуты `<h1>`, но не свойство `frameworkName`. Возможно, вы даже получите предупреждение о неизвестном свойстве `frameworkName` (потому что оно отсутствует в спецификации HTML). Очень удобно!

А теперь присмотритесь к реализации элемента `<div>` (рис. 2.8). Очевидно, она должна рендерить три дочерних элемента класса `HelloWorld`, но текст и атрибуты полученных заголовков (`<h1/>`) должны быть разными. Например, вы передаете значения `id`, `frameworkName` и `title`. Они будут частью класса `HelloWorld`.

Прежде чем реализовывать `<h1/>`, необходимо передать свойства `HelloWorld`. Как это сделать? Эти свойства передаются в объектном литерале во втором аргументе `createElement()` при создании элементов `HelloWorld` в контейнере `<div>`:

```
ReactDOM.render(
  React.createElement(
    'div',
    null,
    React.createElement(HelloWorld, {
      id: 'ember',
      frameworkName: 'Ember.js',
      title: 'A framework for creating ambitious web applications.'}),
```

```
    React.createElement(HelloWorld, {
      id: 'backbone',
      frameworkName: 'Backbone.js',
      title: 'Backbone.js gives structure to web applications...'}),
      React.createElement(HelloWorld, {
    id:'angular',
    frameworkName: 'Angular.js',
    title: 'Superheroic JavaScript MVW Framework'})
  ),
  document.getElementById('content')
)
```

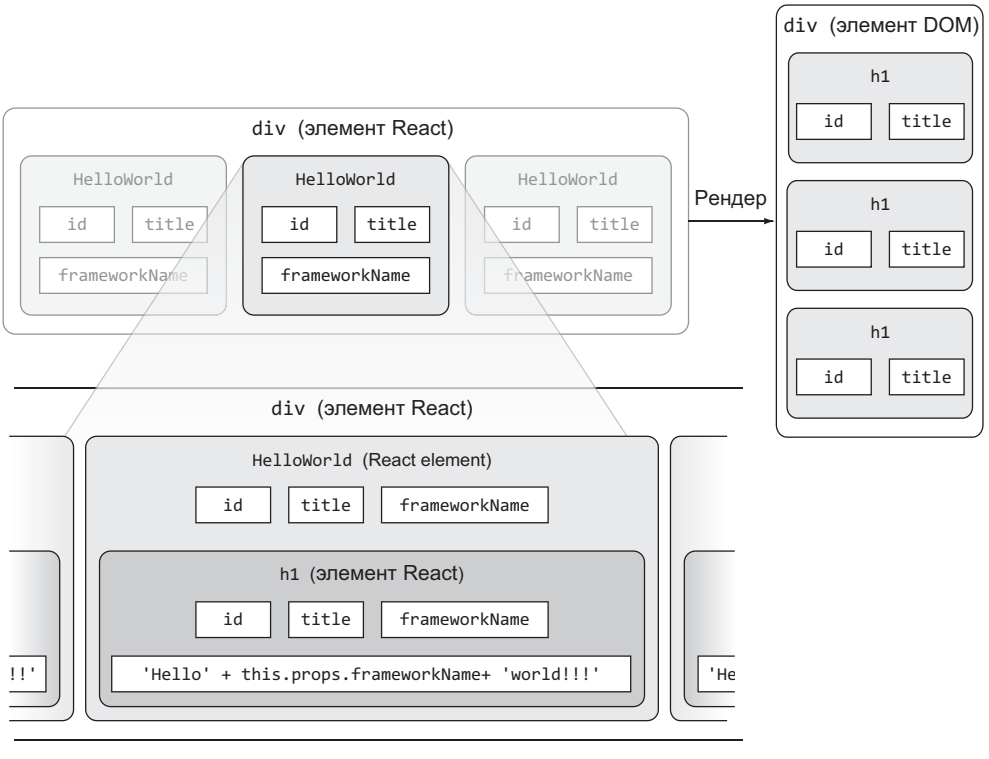


Рис. 2.8. Класс HelloWorld используется три раза для генерирования трех элементов h1 с разными атрибутами и innerHTML

Теперь рассмотрим реализацию компонента `HelloWorld`. Во втором параметре `createElement()` (например, `{id: 'ember'...}`) передается объект, к свойствам которого можно обращаться через объект `this.props` внутри метода `render()` компонента. А значит, к значению `frameworkName` можно обратиться так, как показано в листинге 2.4.

Листинг 2.4. Использование свойства frameworkName в методе render()

```
class HelloWorld extends React.Component {
  render() {
    return React.createElement(
      'h1',
      null,
      'Hello'+this.props.frameworkName + ' world!!!'
    )
  }
}
```

Выполняет конкатенацию (слияние) трех строк: «Hello», «this.props.frameworkName» и «world!!!»

Ключи объекта `this.props` в точности совпадают с ключами объекта, переданного `createElement()` во втором параметре. А именно `this.props` содержит ключи `id`, `frameworkName` и `title`. Количество пар «ключ/значение», которые могут передаваться во втором аргументе `React.createElement()`, неограниченно.

Возможно, вы уже догадались, что все свойства `HelloWorld` могут быть переданы его дочернему элементу `<h1/>`. Это может быть в высшей степени полезно, если вы не знаете, какие свойства передаются компоненту; например, в `HelloWorld` значение атрибута стиля лучше оставить разработчику, который будет создавать экземпляр `HelloWorld`. Это позволяет вам не ограничивать атрибуты для прорисовки в `<h1/>`.

Листинг 2.5. Передача всех свойств из HelloWorld в <h1>

```
class HelloWorld extends React.Component {
  render() {
    return React.createElement(
      'h1',
      this.props,
      'Hello ' + this.props.frameworkName + ' world!!!'
    )
  }
}
```

Передает все свойства дочернему элементу заголовка

Затем три элемента `HelloWorld` рендерятся в `<div>` с идентификатором `content`, как показано в следующем листинге (ch02/hello-js-world/js/script.js) и на рис. 2.9.

Листинг 2.6. Использование свойств, переданных при создании элемента

Выводит свойство frameworkName как текст в <h1>

```
class HelloWorld extends React.Component {
  render() {
    return React.createElement(
      'h1',
      this.props,
      'Hello'+this.props.frameworkName + ' world!!!'
    )
```

Все свойства, переданные HelloWorld при вызове createElement, передаются этому элементу <h1>

```
  }
}
ReactDOM.render(
  React.createElement(
    'div',
    null,
    React.createElement(HelloWorld, {
      id: 'ember', 3((CO5-3))
      frameworkName: 'Ember.js',
      title: 'A framework for creating ambitious web applications.'}),
    React.createElement(HelloWorld, {
      id: 'backbone',
      frameworkName: 'Backbone.js',
      title: 'Backbone.js gives structure to web applications...'}),
    React.createElement(HelloWorld, {
      id: 'angular',
      frameworkName: 'Angular.js',
      title: 'Superheroic JavaScript MVW Framework'})
  ),
  document.getElementById('content')
)
```

id и title соответствуют стандартным атрибутам HTML для <h1> и рендерятся как эти атрибуты

frameworkName не является стандартным атрибутом HTML для <h1>, поэтому он не будет рендериться, если не принять каких-то специальных мер

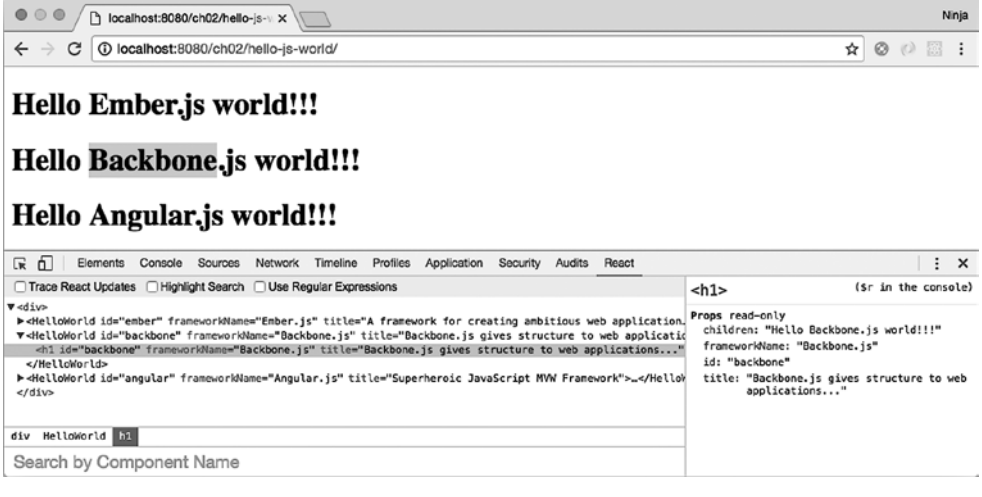


Рис. 2.9. Результат многократного использования HelloWorld с разными свойствами для рендера трех разных заголовков

Как обычно, этот код можно выполнить с локальным веб-сервером HTTP. В результате повторного использования класса компонента HelloWorld вы получаете три разных заголовка (рис. 2.9).

Мы использовали `this.props` для генерирования разного текста заголовков. Таким образом, практически весь код был использован заново, а вы стали повелителем классов компонента `HelloWorld`!

В этой главе было рассмотрено несколько модификаций приложения Hello World. Да, я знаю — это все равно старое, доброе и малоинтересное приложение. Но начиная с малого, мы закладываем прочный фундамент для будущих, более сложных тем. Поверьте, с классами компонентов можно добиться гораздо большего.

Очень важно знать, как React работает в обычных событиях JavaScript, если вы (как и многие программисты React) собираетесь использовать JSX. Ведь в конечном итоге в браузерах все равно выполняется стандартный код JS, а время от времени появляется необходимость понимать результаты транспиляции JSX в JS. Забегая вперед, скажу, что мы *будем* использовать JSX, — эта тема рассматривается в следующей главе.

2.4. Вопросы

1. Какие из следующих конструкций могут использоваться для создания классов компонентов React: `createComponent()`, `createElement()`, `class NAME extends React.Component`, `class NAME extends React.Class`?
2. Какой из следующих атрибутов или методов компонента React является единственным обязательным: `function`, `return`, `name`, `render`, `class`?
3. Какие из следующих конструкций могут использоваться для обращения к свойству `url` компонента: `this.properties.url`, `this.data.url`, `this.props.url`, `url`?
4. Свойства React являются неизменяемыми в контексте текущего компонента. Да или нет?
5. Классы компонентов React позволяют разработчикам создавать пользовательские интерфейсы, пригодные для повторного использования. Да или нет?

2.5. Итоги

- Для создания вложенных элементов React используются третий, четвертый и так далее аргументы `createElement()`.
- Элементы строятся на базе нестандартных классов компонентов.
- Для изменения генерируемых элементов используются свойства.
- Свойства могут передаваться дочерним элементам.
- Чтобы использовать компонентную архитектуру (одна из особенностей React), вы создаете компоненты.

1. `class NAME extends React.Component`, потому что `React.Class` не существует, а остальные варианты завершатся неудачей из-за ошибки `ReferenceError` (не определено).
2. `render()`, потому что это единственный обязательный метод; также потому что `function`, `return`, `render` и `class` недействительны, а `name` обязательным не является.
3. `this.props.url`, потому что только `this.props` предоставляет объект свойств.
4. Да. Изменить свойство невозможно.
5. Да. Разработчики используют новые компоненты для создания пользовательских интерфейсов, пригодных для повторного использования.

2.6. Ответы

3 Знакомство с JSX

Посмотрите вступительный видеоролик к этой главе, отсканировав QR-код или перейдя по ссылке http://reactquickly.co/videos/ch03.

ЭТА ГЛАВА ОХВАТЫВАЕТ СЛЕДУЮЩИЕ ТЕМЫ:

- Понимание JSX и его преимуществ.
- Настройка транспилятора JSX с Babel.
- Потенциальные проблемы React и JSX.

Итак, пора познакомиться с JSX! На мой взгляд, это одна из самых замечательных особенностей React — и одна из самых неоднозначных, по мнению некоторых разработчиков, с которыми я общался (они, как и следовало ожидать, еще не построили ни одного крупного проекта в React).

До сих пор мы обсуждали, как создавать элементы и компоненты, чтобы вы могли применять нестандартные элементы и лучше организовать свой пользовательский интерфейс. Для создания элементов React использовался код JavaScript (вместо прямой работы с HTML). Однако здесь возникает одна проблема. Взгляните на следующий код и попробуйте понять, что здесь происходит:

```
render() {
  return React.createElement(
    'div',
    { style: this.styles },
    React.createElement(
      'p',
      null,
      React.createElement(
        reactRouter.Link,
        { to: this.props.returnTo },
```

```
        'Back'
      )
    ),
    this.props.children
  );
}
```

Вы смогли с ходу определить, что здесь создаются три элемента, что они вложены, что в коде используется компонент из React Router? Насколько хорошо читался этот код по сравнению со стандартной разметкой HTML? Назовете ли вы этот код выразительным? Команда React согласна с тем, что читать (да и вводить, если уж на то пошло) целую кучу команд `React.createElement()` совершенно неинтересно. Решением этой проблемы стал язык JSX.

ПРИМЕЧАНИЕ Исходный код примеров этой главы доступен по адресам www.manning.com/books/react-quickly и https://github.com/azat-co/react-quickly/tree/master/ch03 (в папке ch03 репозитория GitHub https://github.com/azat-co/react-quickly). Также некоторые демонстрационные примеры доступны по адресу http://reactquickly.co/demos.

3.1. Что такое JSX и чем он хорош?

JSX — расширение JavaScript, предоставляющее синтаксический сахар для вызовов функций и построения объектов, особенно `React.createElement()`. На первый взгляд код JSX может показаться чем-то вроде шаблонизатора или HTML, но это не так. JSX строит элементы React, позволяя вам пользоваться всей мощью JavaScript.

JSX — отличный инструмент для записи компонентов React. Его основные преимущества:

- *Улучшенное восприятие со стороны разработчика* (DX, Developer Experience) — код проще читается из-за его выразительности, обусловленной XML-подобным синтаксисом, лучше подходящим для представления вложенных декларативных структур.
- *Повышение эффективности работы команд* — неопытным разработчикам (например, веб-дизайнерам) проще изменять этот код, потому что JSX напоминает разметку HTML, которая им уже знакома.
- *Снижение нагрузки на пальцы и количества синтаксических ошибок* — разработчикам приходится набирать меньше кода, а это означает, что они сделают меньше ошибок и им придется выполнять меньше монотонной работы.

И хотя язык JSX не обязателен, он хорошо сочетается с React, и создатели React (и я вместе с ними) рекомендуют использовать его. На официальной странице «Introducing JSX»[1] сказано: «Мы рекомендуем использовать [JSX] с React».

[1] https://facebook.github.io/react/docs/introducing-jsx.html.

Чтобы продемонстрировать выразительность JSX, я приведу код создания `HelloWorld` и элемента ссылки:

```
<div>
  <HelloWorld/>
  <br/>
  <a href="http://webapplog.com">Great JS Resources</a>
</div>
```

Этот фрагмент аналогичен следующему фрагменту JavaScript:

```
React.createElement(
  "div",
  null,
  React.createElement(HelloWorld, null),
  React.createElement("br", null),
  React.createElement(
    "a",
    { href: "http://webapplog.com" },
    "Great JS Resources"
  )
)
```

А если вы воспользуетесь Babel v6 (один из инструментов для работы с JSX; подробнее о Babel я расскажу через несколько страниц), код JS принимает следующий вид:

```
"use strict";

React.createElement(
  "div",
  null,
  " ",
  React.createElement(HelloWorld, null),
  " ",
  React.createElement("br", null),
  " ",
  React.createElement(
    "a",
    { href: "http://webapplog.com" },
    "Great JS Resources"
  ),
  " "
);
```

Фактически JSX представляет собой мини-язык с XML-подобным синтаксисом; однако этот язык изменил подход к написанию UI-компонентов. Ранее разработчики писали разметку HTML — и код JS для контроллеров и представлений — в MVC-подобном стиле, переключаясь между файлами. Такой подход был обусловлен разделением обязанностей на ранней стадии развития технологий. Он хорошо работал, пока веб-страницы состояли из статической разметки HTML, небольшого количества CSS и крошечного фрагмента JS для создания мигающего текста.

Времена изменились. В наши дни строятся высокоинтерактивные пользовательские интерфейсы, а JS и HTML тесно связаны друг с другом для реализации различных аспектов функциональности. React исправляет нарушенный принцип разделения обязанностей (SoC), объединяя описание пользовательского интерфейса и логики JS; с JSX же код походит на HTML, он проще читается и пишется. Только по одной этой причине я бы использовал React и JSX ради нового подхода к написанию пользовательских интерфейсов.

Код JSX компилируется различными преобразователями (инструментами) в стандартный ECMAScript (рис. 3.1). Вероятно, вы знаете, что JavaScript также является стандартом ECMAScript, но JSX в эту спецификацию не входит и не обладает никакой определенной семантикой.

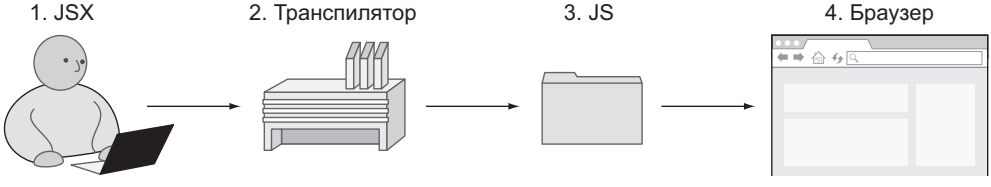


Рис. 3.1. JSX транспилируется в стандартный код JavaScript

ПРИМЕЧАНИЕ Согласно https://en.wikipedia.org/wiki/Source-to-source_compiler, «Транскомпилятор, или транспилятор, — разновидность компилятора, которая получает исходный код программы, написанный на одном языке программирования, и генерирует эквивалентный исходный код на другом языке программирования».

Возникает резонный вопрос: «Зачем мне возиться с JSX?» Хороший вопрос. Если учесть, насколько противоестественно выглядит код JSX на первых порах, неудивительно, что многие разработчики отвернулись от этой замечательной технологии. Например, в следующем фрагменте JSX в коде JavaScript встречаются угловые скобки, которые выглядят очень странно:

```
ReactDOM.render(<h1>Hello</h1>, document.getElementById('content'))
```

Одна из самых замечательных особенностей JSX — сокращенные формы записи для `React.createElement(NAME, ...)`. Вместо того чтобы снова и снова записывать этот вызов функции, достаточно использовать `<NAME/>`. И как я говорил ранее, чем меньше символов вы вводите, тем меньше ошибок совершите. С JSX восприятие разработчика не менее важно, чем восприятие пользователя (UX, User Experience).

Главная причина для использования JSX заключается в том, что, по мнению многих людей, код в угловых скобках (`< >`) читается лучше, чем код с большим количеством команд `React.createElement()` (даже с синонимами). А когда вы

привыкнете рассматривать <NAME/> не как XML, а как синоним для кода JavaScript, вся субъективная странность синтаксиса JSX исчезнет. Знание и использование JSX может сильно повлиять на разработку компонентов React и, соответственно, приложений на базе React.

ДРУГИЕ ВАРИАНТЫ СОКРАЩЕННОЙ ЗАПИСИ

По правде говоря, для предотвращения ввода объемистых вызовов `React.createElement()` у JSX есть несколько альтернатив. Одна из них — использование синонима React.DOM.*. Например, вместо того чтобы создавать элемент `<h1/>` вызовом

```
React.createElement('h1', null, 'Hey')
```

можно обойтись следующим вызовом, который занимает меньше места и быстрее вводится:

```
React.DOM.h1(null, 'Hey')
```

Вы получаете доступ ко всем стандартным элементам HTML в объекте React.DOM, который можно проанализировать как любой другой объект.

```
console.log(React.DOM)
```

Также можно ввести React.DOM и нажать клавишу Enter в консоли Chrome DevTools. (Учтите, что React.DOM и ReactDOM – два совершенно разных объекта, не путайте их и не пытайтесь использовать один вместо другого).

Другая альтернатива, рекомендованная в официальной документации React для ситуаций, в которых использование JSX непрактично (например, при отсутствии процесса сборки), основана на использовании короткой переменной. Например, переменную E можно создать следующим образом:

```
const E = React.createElement
E('h1', null, 'Hey')
```

Как я упоминал ранее, код JSX необходимо транспилировать в стандартный код JavaScript перед тем, как он будет выполнен в браузере. Различные способы будут рассмотрены в разделе 3.3; там же будет показан рекомендованный способ.

3.2. Логика JSX

А теперь посмотрим, как работать с JSX. Вы можете прочитать этот раздел и оставить в нем закладку на будущее, или (если вы предпочитаете выполнить некоторые примеры на своем компьютере) у вас есть следующие варианты:

- Установите транспилятор JSX с Babel на своем компьютере, как показано в разделе 3.3.
- Воспользуйтесь сетевым сервисом Babel REPL (https://babeljs.io/repl), который транспилирует JSX в JavaScript в браузере.

Выбор за вами. Я рекомендую сначала ознакомиться с основными концепциями JSX, а уже потом переходить к настройке Babel на компьютере.

3.2.1. Создание элементов с JSX

Объекты `ReactElement` в JSX создаются достаточно прямолинейно. Например, вместо следующего кода JavaScript (где `name` является строкой — `h1` — или классом компонента — `HelloWorld`):

```
React.createElement(
  name,
  {key1: value1, key2: value2, ...},
  child1, child2, child3, ..., childN
  )
```

можно написать следующий фрагмент JSX:

```
<name key1=value1 key2=value2 ...>
  <child1/>
  <child2/>
  <child3/>
  ...
  <childN/>
</name>
```

В коде JSX атрибуты и их значения (например, `key1=value1`) берутся из второго аргумента `createElement()`. Работа со свойствами будет более подробно рассмотрена позднее, а пока рассмотрим пример элемента JSX без свойств. В листинге 3.1 приведена хорошо знакомая программа Hello world на JavaScript (ch03/hello-world/index.html).

Листинг 3.1. Программа Hello world на JavaScript

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

Версия на JSX получается намного более компактной (ch03/hello-world-jsx/js/script.jsx).

Листинг 3.2. Программа Hello world на JSX

```
ReactDOM.render(
  <h1>Hello world!</h1>,
  document.getElementById('content')
)
```

Также объекты, созданные в синтаксисе JSX, могут сохраняться в переменных, потому что JSX является всего лишь синтаксическим улучшением для `React.createElement()`. В следующем примере ссылка на объект `Element` сохраняется в переменной:

```
let helloWorldReactElement = <h1>Hello world!</h1>
ReactDOM.render(
  helloWorldReactElement,
  document.getElementById('content')
)
```

3.2.2. Работа с JSX в компонентах

В предыдущем примере используется тег JSX `<h1>`, который также является стандартным тегом HTML. При работе с компонентами применяется такой же синтаксис. Единственное различие заключается в том, что имя класса компонента должно начинаться с буквы верхнего регистра, как в `<HelloWorld/>`.

Ниже приведена улучшенная версия Hello World, переписанная на JSX. В этом случае создается новый класс компонента, а JSX используется для создания элемента на его базе.

Листинг 3.3. Создание класса HelloWorld в JSX

```
class HelloWorld extends React.Component {
  render() {
    return (
      <div>
        <h1>1. Hello world!</h1>
        <h1>2. Hello world!</h1>
      </div>
    )
  }
}
ReactDOM.render(
  <HelloWorld/>,
  document.getElementById('content')
)
```

Читается ли листинг 3.3 проще, чем следующий код JavaScript?

```
class HelloWorld extends React.Component {
  render() {
    return React.createElement('div',
      null,
      React.createElement('h1', null, '1. Hello world!'),
      React.createElement('h1', null, '2. Hello world!'))
  }
}
ReactDOM.render(
```

```
  React.createElement(HelloWorld, null),
  document.getElementById('content')
)
```

ПРИМЕЧАНИЕ Как упоминалось ранее, угловые скобки в коде JavaScript могут показаться странными для опытного разработчика JavaScript. Когда я впервые увидел их, у меня голова пошла кругом, потому что я много лет привыкал с ходу обнаруживать синтаксические ошибки JS! Скобки — едва ли не главная претензия к JSX, которые мне приходится слышать, вот почему мы займемся JSX почти в начале книги — чтобы у вас было как можно больше опыта работы с ним.

Обратите внимание на круглые скобки после `return` в коде JSX из листинга 3.3; их присутствие обязательно, если в одной строке после `return` ничего больше нет. Например, если вы размещаете элемент верхнего уровня `<div>` в новой строке, его необходимо заключить в круглые скобки `()`. В противном случае JavaScript воспримет это как признак завершения пустой команды `return`. Этот стиль выглядит так:

```
render() {
  return (
    <div>
    </div>
  )
}
```

Также можно начать элемент верхнего уровня в одной строке с `return`, тогда круглые скобки `()` перестают быть обязательными. Например, следующий фрагмент тоже допустим:

```
render() {
  return <div>
  </div>
}
```

Недостатком второго варианта можно считать то, что открывающий тег `<div>` становится менее заметным: его легко упустить в коде[1]. Выбор за вами. Я буду применять в книге оба стиля, чтобы вы более полно представляли ситуацию.

3.2.3. Вывод переменных в JSX

Написанные вами компоненты должны быть достаточно умными, чтобы изменять свое представление в зависимости от некоторого кода. Например, компонент для

[1] За подробностями этого поведения в JavaScript обращайтесь к статьям Джеймса Нельсона (James Nelson) «Why Use Parenthesis [sic] on JavaScript Return Statements?», 11 августа 2016 г. (http://jamesknelson.com/javascript-return-parenthesis), и «Automated Semicolon Insertion», Annotated ECMAScript 5.1, http://es5.github.io/#x7.9.

отображения даты/времени должен отображать текущую дату и время, а не жестко фиксированное значение.

При работе с React только с JavaScript приходится прибегать к конкатенации (+) или, если вы используете ES6+/ES2015, — к строковым шаблонам в обратных апострофах, содержащим конструкцию `${varName}`, где `varName` — имя переменной. Согласно спецификации, данная возможность официально именуется *шаблонным литералом* (template literal[1]). Например, чтобы использовать свойство в тексте компонента `DateTimeNow` в стандартном JavaScript с React, можно написать следующий код:

```
class DateTimeNow extends React.Component {
  render() {
    let dateTimeNow = new Date().toLocaleString()
    return React.createElement(
      'span',
      null,
      `Current date and time is ${dateTimeNow}.`
    )
  }
}
```

С другой стороны, в JSX для динамического вывода переменных используются фигурные скобки {}, с которыми код становится существенно более компактным:

```
class DateTimeNow extends React.Component {
  render() {
    let dateTimeNow = new Date().toLocaleString()
    return <span>Current date and time is {dateTimeNow}.</span>
    )
  }
}
```

Переменные также могут быть свойствами, а не локально определенными переменными:

```
<span>Hello {this.props.userName}, your current date and time is
➥ {dateTimeNow}.</span>
```

Более того, внутри {} могут выполняться любые выражения JavaScript или любой код JS. Например, выводимую дату можно отформатировать:

```
<p>Current time in your locale is
➥ {new Date(Date.now()).toLocaleTimeString()}</p>
```

А теперь класс `HelloWorld` можно переписать на JSX с использованием динамических данных, которые JSX хранит в переменной (ch03/hello-world-class-jsx).

---

[1] «Template Literals,» ECMAScript 2015 Language Specification, июнь 2015 г., http://mng.bz/i8Bw.

Листинг 3.4. Вывод переменных в JSX

```
let helloWorldReactElement = <h1>Hello world!</h1>
class HelloWorld extends React.Component {
  render() {
    return <div>
      {helloWorldReactElement}
      {helloWorldReactElement}
    </div>
  }
}
ReactDOM.render(
  <HelloWorld/>,
  document.getElementById('content')
)
```

А теперь разберемся, как работать со свойствами в JSX.

3.2.4. Работа со свойствами в JSX

Эта тема уже была затронута ранее, когда я представлял JSX: свойства элементов определяются с использованием синтаксиса атрибутов, а именно: запись `ключ1=значение1 ключ2=значение2...` в теге JSX используется для определения как атрибутов HTML, так и свойств компонента React. В этом отношении она напоминает синтаксис атрибутов в HTML/XML.

Иначе говоря, если вам понадобится передать свойства, запишите их в JSX так же, как вы бы это сделали в обычной разметке HTML. Кроме того, для рендера стандартных атрибутов HTML задаются значения свойств элементов (см. раздел 2.3). Например, следующий код задает стандартный атрибут HTML `href` для элемента `<a>`:

```
ReactDOM.render((
  <div>
    <a href="http://reactquickly.co">Time for React?</a>
    <DateTimeNow userName='Azat'/>
  </div>
  ),
  document.getElementById('content')
)
```

Рендерит стандартный атрибут HTML href

Определяет значение для свойства userName

Фиксированным значениям атрибутов не хватает гибкости. Если вы хотите заново использовать компонент ссылки, атрибут `href` должен изменяться, чтобы он каждый раз отражал новый адрес. Это называется *динамическим присваиванием значений*, в отличие от их жесткой фиксации в программе. Сейчас мы сделаем следующий шаг и рассмотрим компонент, который может использовать динамически сгенерированные значения для атрибутов. Эти значения можно брать из свойств компонентов (`this.props`). После этого все просто. От вас потребуется только использовать

фигурные скобки ({}) внутри угловых скобок (<>) для передачи динамических значений свойств элементам.

Допустим, вы строите компонент, который будет использоваться для ссылок на учетные записи пользователей. Значения `href` и `title` должны быть разными, поэтому они не могут жестко фиксироваться в коде. Динамический компонент `ProfileLink` рендерит ссылку `<a>`, используя свойства `url` и `label` для `href` и `title` соответственно. В `ProfileLink` для передачи свойств `<a>` используются `{}`:

```
class ProfileLink extends React.Component {
  render() {
    return <a href={this.props.url}
      title={this.props.label}
      target="_blank">Profile
    </a>
  }
}
```

Откуда берутся значения свойств? Они определяются при создании `ProfileLink`, то есть в компоненте, который создает `ProfileLink` (в его родителе). Например, так передаются значения `url` и `label` при создании экземпляра `ProfileLink`, в результате чего генерируется тег `<a>` со следующими значениями:

```
<ProfileLink url='/users/azat' label='Profile for Azat'/>
```

В предыдущей главе говорилось о том, что при рендере стандартных элементов (`<h>`, `<p>`, `<div>`, `<a>` и т. д.) React рендерит все атрибуты из спецификации HTML и опускает все остальные атрибуты, которые в спецификацию не входят. Это не какая-то особенность JSX, а поведение React.

Но иногда бывает нужно добавить нестандартные данные в виде атрибута. Допустим, имеется элемент списка; он содержит информацию, которая критична для вашего приложения, но не нужна пользователям. Обычно в таких случаях подобная информация размещается в элементе DOM в качестве атрибута. В следующем примере используются атрибуты `react-is-awesome` и `id`:

```
<li react-is-awesome="true" id="320">React is awesome!</li>
```

Хранение данных в нестандартных атрибутах HTML в DOM обычно считается антипаттерном, потому что модель DOM не должна превращаться в базу данных или хранилище данных клиентской части. Чтение данных из DOM выполняется медленнее, чем чтение из виртуального хранилища/памяти.

В тех случаях, когда данные требуется сохранить в атрибутах элементов, а вы используете JSX, необходимо использовать префикс `data-NAME`. Например, чтобы отрендерить элемент `<li>` со значением `this.reactIsAwesome` в атрибуте, можно использовать следующую запись:

```
<li data-react-is-awesome={this.reactIsAwesome}>React is awesome!</li>
```

Допустим, значение `this.reactIsAwesome` истинно. Тогда сгенерированная разметка HTML выглядит так:

```
<li data-react-is-awesome="true">React is awesome!</li>
```

Но если вы попытаетесь передать нестандартный атрибут HTML стандартному элементу HTML, этот атрибут рендериться не будет (см. раздел 2.3). Например, фрагмент

```
<li react-is-awesome={this.reactIsAwesome}>React is orange</li>
```

и фрагмент

```
<li reactIsAwesome={this.reactIsAwesome}>React is orange</li>
```

генерируют только следующую разметку:

```
<li>React is orange</li>
```

Очевидно, поскольку нестандартные элементы (классы компонентов) не имеют встроенных средств рендеринга и зависят от стандартных элементов HTML или других нестандартных элементов, проблема использования `data-` для них не важна. Они получают все атрибуты в виде свойств в `this.props`.

Раз уж речь зашла о классах компонентов, вспомним код из Hello World (раздел 2.3), написанный на стандартном JavaScript:

```
class HelloWorld extends React.Component {
  render() {
    return React.createElement(
      'h1',
      this.props,
      'Hello'+this.props.frameworkName + ' world!!!'
    )
  }
}
```

В компонентах `HelloWorld` свойства передаются `<h1>` независимо от их состава. Как сделать это в JSX? Передавать каждое свойство по отдельности не хочется, потому что это увеличит объем кода, а когда свойство потребуется изменить, у вас появится сильно связанный код, который тоже придется обновлять. Представьте, что каждое свойство будет передаваться вручную, — а если оно должно пройти через два или три уровня компонентов? Получается антипаттерн. Не делайте так:

```
class HelloWorld extends React.Component {
  render() {
    return <h1 title={this.props.title} id={this.props.id}>
      Hello {this.props.frameworkName} world!!!
    </h1>
  }
}
```

Не передавайте свойства по отдельности, если вы намерены передать их *все*; в JSX есть специальный синтаксис, напоминающий многоточие `...`; он продемонстрирован в следующем листинге (ch03/jsx/hello-js-world-jsx).

Листинг 3.5. Работа со свойствами

```
class HelloWorld extends React.Component {
  render() {
    return <h1 {...this.properties}>
      Hello {this.props.frameworkName} world!!!
    </h1>
  }
}

ReactDOM.render(
  <div>
    <HelloWorld
      id='ember'
      frameworkName='Ember.js'
      title='A framework for creating ambitious web applications.'/>,
    <HelloWorld
      id='backbone'
      frameworkName= 'Backbone.js'
      title= 'Backbone.js gives structure to web applications...'/>
    <HelloWorld
      id= 'angular'
      frameworkName= 'Angular.js'
      title= 'Superheroic JavaScript MVW Framework'/>
  </div>,
  document.getElementById('content')
)
```

С синтаксисом `{...this.props}` можно передать дочернему элементу все свойства. Остальной код представляет собой просто преобразованный для JSX пример из раздела 2.3.

МНОГОТОЧИЯ В ES6+/ES2015+: ОСТАТОК, РАСШИРЕНИЕ И ДЕСТРУКТУРИЗАЦИЯ

Раз уж речь зашла о многоточиях, в ES6+ существуют похожие операторы деструктуризации (destructuring), расширения (spread) и остатка (rest). Собственно, это одна из причин для использования многоточий в React JSX!

Если вы когда-либо писали или использовали функцию JavaScript с переменным или неограниченным количеством аргументов, то объект arguments вам уже знаком. Этот объект содержит все параметры, переданные функции. К сожалению, объект arguments не является настоящим массивом. Его необходимо преобразовать в массив, чтобы явно использовать такие функции, как `sort()` и `map()`. Например, следующая функция request преобразует аргументы вызовом `call()`:

```
function request(url, options, callback) {
  var args = Array.prototype.slice.call(arguments, request.length)
  var url = args[0]
  var callback = args[2]
  // ...
}
```

Существует ли в ES6 более удобный способ работать с неопределенным количеством аргументов как с массивом? Да! Это синтаксис остатка параметров, определяемый многоточием (`...`). Например, вот сигнатура функции ES6 с параметром `callbacks`, в который заносится массив (настоящий массив, не псевдомассив аргументов) с остатком параметров[1]:

```
function(url, options, ...callbacks) {
  var callback1 = callbacks[0]
  var callback2 = callbacks[1]
  // ...
}
```

Параметр-остаток можно деструктуризировать, то есть разложить на отдельные переменные:

```
function(url, options, ...[error, success]) {
  if (!url) return error(new Error('ooops'))
  // ...
  success(data)
}
```

Как насчет расширения? В двух словах, оно позволяет расширять аргументы или переменные в следующих местах:

- *Вызовы функций* — например, метод `push()`: `arr1.push(…arr2)`
- *Литералы массивов* — например, `array2 = [...array1, x, y, z]`
- *Вызовы функции* `new` *(конструкторы)* — например, `var d = new Date(...dates)`

Если вы хотели использовать массив в аргументе функции в ES5, вам приходилось использовать функцию `apply()`:

```
function request(url, options, callback) {
  // ...
}
var requestArgs = ['http://azat.co', {...}, function(){...}]
request.apply(null, requestArgs)
```

В ES6 можно воспользоваться параметром-расширением, который по синтаксису похож на параметр-остаток и использует многоточие (`...`):

[1] В массиве первым идет параметр, не имеющий имени: например, `callback` имеет индекс 0, а не 2, как в массиве arguments в ES5. Кроме того, размещение других именованных аргументов после параметра-остатка считается синтаксической ошибкой.

```
function request(url, options, callback) {
  // ...
}
var requestArgs = ['http://azat.co', {...}, function(){...}]
request(...requestArgs)
```

Синтаксис оператора расширения напоминает синтаксис параметра-остатка, но остаток используется в определении/объявлении функции, а расширение — в вызовах и литералах.

Эти операторы избавляют вас от необходимости вводить лишние строки императивного кода, поэтому знать и использовать их будет полезно.

3.2.5. Создание методов компонентов React

Разработчик может писать для своих приложений любые методы компонентов, потому что компонент React является классом. Например, можно создать вспомогательный метод `getUrl()`:

```
class Content extends React.Component {
  getUrl() {
    return 'http://webapplog.com'
  }
  render() {
    ...
  }
}
```

Метод `getUrl()` сложностью не отличается, но вы наверняка поняли главное: вы можете создавать произвольные методы по своему усмотрению, не только `render()`. Метод `getUrl()` позволяет абстрагировать URL-адрес для сервера API. Вспомогательные методы могут содержать логику, пригодную для повторного использования, и вы можете вызывать их в любом месте с другими методами компонента, включая `render()`.

Если вы захотите вывести возвращаемое значение нестандартного метода в JSX, используйте `{}` как и с переменными (см. листинг 3.6, ch03/method/jsx/scrch03/meipt.jsx). В данном случае вспомогательный метод вызывается в `render`, а возвращаемые значения метода будет использованы в представлении. Не забудьте, что метод должен вызываться с `()`.

Листинг 3.6. Вызов метода компонента для получения URL

```
class Content extends React.Component {
  getUrl() {
    return 'http://webapplog.com'
  }
  render() {
    return (
```

```
      <div>
        <p>Your REST API URL is:
          <a href={this.getUrl()}>  ◄— Вызывает метод класса в фигурных скобках
            {this.getUrl()}
          </a>
        </p>
      </div>
    )
  }
}
...
```

И снова — методы компонентов можно вызывать напрямую из `{}` и JSX. Например, при использовании `{this.getUrl()}` в листинге 3.6 вы увидите возвращаемое значение http://webapplog.com в ссылке из абзаца `<p>` (рис. 3.2).

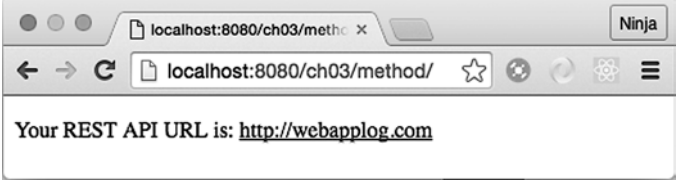


Рис. 3.2. Результаты генерирования ссылки со значением, полученным от метода

В общем, с методами компонентов все должно быть понятно. Я прошу прощения, если этот раздел показался вам слишком банальным; эти методы играют важную роль как основа для обработчиков событий React.

3.2.6. if/else в JSX

Как и в случае с динамическими переменными, разработчики должны строить свои компоненты таким образом, чтобы компоненты могли изменять представления в зависимости от результатов условий `if/else`.

Начнем с простого примера, в котором рендерятся элементы класса компонента; элементы зависят от условия. Например, текст ссылки и URL-адрес определяются значением `user.session`. В обычном коде JS решение могло бы выглядеть так:

```
...
render() {
  if (user.session)
    return React.createElement('a', {href: '/logout'}, 'Logout')
  else
    return React.createElement('a', {href: '/login'}, 'Login')
  }
...
```

С JSX можно действовать аналогичным образом и переписать код в следующем виде:

```
...
render() {
  if (this.props.user.session)
    return <a href="/logout">Logout</a>
  else
    return <a href="/login">Login</a>
  }
...
```

Предположим, есть и другие элементы, например контейнер `<div>`. В этом случае в простом коде JS придется создать переменную, использовать выражение или тернарный оператор (известный среди молодых разработчиков JavaScript под названием «оператор Элвиса»; см. http://mng.bz/92Zg), потому что внутри `createElement()` элемента `<div>` нельзя использовать условие `if`.

ТЕРНАРНЫЕ ОПЕРАТОРЫ

В следующем тернарном условии, если значение `userAuth` истинно, то `msg` присваивается строка `'welcome'`. В противном случае будет присвоено значение 'restricted':

```
let msg = (userAuth) ? 'welcome' : 'restricted'
```

Эта команда эквивалентна следующей:

```
let session = ''
if (userAuth) {
  session = 'welcome'
}else{
  session = 'restricted'
}
```

В некоторых случаях тернарный оператор (`?`) представляет собой более короткую версию if/else. Но между ними существует серьезное различие, если вы попытаетесь использовать тернарный оператор в выражении (где он должен вернуть значение). Следующий код является допустимым с точки зрения JS:

```
let msg = (userAuth) ? 'welcome' : 'restricted'
```

Но if/else не сработает, потому что это не выражение, а команда:

```
let msg = if (userAuth) {'welcome'} else {'restricted'} // Недопустимо
```

Эту особенность тернарного оператора можно использовать для получения значения от него во время выполнения в JSX.

Чтобы продемонстрировать три разных стиля (переменная, выражение и тернарный оператор), рассмотрим следующий код JavaScript перед его преобразованием в JSX:

```
// Способ 1: Переменная
render() {
  let link
  if (this.props.user.session)
    link = React.createElement('a', {href: '/logout'}, 'Logout')
  else
    link = React.createElement('a', {href: '/login'}, 'Login')
  return React.createElement('div', null, link)  <-- Использует переменную link
}
// Способ 2: Выражение
render() {
  let link = (sessionFlag) => {  <-- Создает выражение
    if (sessionFlag)
      return React.createElement('a', {href: '/logout'}, 'Logout')
    else
      return React.createElement('a', {href: '/login'}, 'Login')
  }
  return React.createElement('div', null, link(this.props.user.session))
}
// Способ 3: Тернарный оператор
render() {
  return React.createElement('div', null,
    (this.props.user.session) ? React.createElement('a', {href: '/logout'},
      ➥ 'Logout') : React.createElement('a', {href: '/login'}, 'Login')
  )
}
```

Использует тернарный оператор

Неплохо, но неуклюже. Согласны? С JSX синтаксис `{}` может выводить значения переменных и выполнять код JS. Попробуем улучшить синтаксис:

```
// Способ 1: Переменная
render() {
  let link
  if (this.props.user.session)
    link = <a href='/logout'>Logout</a>
  else
    link = <a href='/login'>Login</a>
  return <div>{link}</div>
}
// Способ 2: Выражение
render() {
  let link = (sessionFlag) => {
    if (sessionFlag)
      return <a href='/logout'>Logout</a>
    else
      return <a href='/login'>Login</a>
  }
  return <div>{link(this.props.user.session)}</div>
```

```
}
// Способ 3: Тернарный оператор
render() {
  return <div>
    {(this.props.user.session) ? <a href='/logout'>Logout</a> :
    ➥ <a href='/login'>Login</a>}
  </div>
}
```

Более внимательно присмотревшись к примеру с выражением/функцией (способ 2: функция вне JSX перед `return`), вы заметите альтернативу. Ту же функцию можно определить с выражением немедленно вызываемой функции, или IIFE (http://mng.bz/387u) внутри JSX. Это позволяет вам избежать создания лишней переменной (такой, как `link`) и отработать `if/else` во время выполнения:

```
render() {
  return <div>{
    (sessionFlag) => {  ◄── Определяет IIFE
      if (sessionFlag)
        return <a href='/logout'>Logout</a>
      else
        return <a href='/login'>Login</a>
    }(this.props.user.session)  ◄── Вызывает IIFE с параметром
  }</div>
}
```

Кроме того, те же принципы могут применяться для рендера не только целых элементов (`<a>` в этих примерах), но также текста и значений свойств. Для этого необходимо лишь использовать один из продемонстрированных способов в фигурных скобках. Например, можно расширить URL и текст без дублирования кода для создания элемента. Лично мне этот способ нравится больше остальных, потому что я могу использовать один тег `<a>`:

```
                                  Создает локальную переменную для хранения сеансового
                                  логического значения (для уменьшения объема кода
                                  и повышения быстродействия)
  render() {                                                    ◄┐
    let sessionFlag = this.props.user.session                   ◄┘
    return <div>
      <a href={(sessionFlag)?'/logout':'/login'}>
        {(sessionFlag)?'Logout':'Login'}  ◄── Использует тернарный оператор
      </a>                                    для рендера другого текста
    </div>
  }

  Использует тернарный оператор для рендера
  разных URL в зависимости от значения sessionFlag
```

Как видите, в отличие от шаблонизаторов, в JSX не существует специального синтаксиса для таких условий — вы используете обычный код JavaScript. Чаще всего в таких ситуациях используется тернарный оператор, потому что это один из

самых компактных стилей. Подведем итог: при реализации логики `if/else` в JSX доступны следующие варианты:

- Переменная, определяемая за пределами JSX (перед `return`) и выводимая конструкцией `{}` в JSX.
- Выражение (функция, возвращающая значение), определяемое вне JSX (перед `return`) и выполняемое в `{}` в JSX.
- Условный тернарный оператор.
- IIFE в JSX.

Это основное правило в том, что касается условий и JSX: используйте `if/else` за пределами JSX (перед `return`) для генерирования значения, которое выводится в JSX в `{}`. Другой способ — обойтись без переменной и вывести результаты тернарного оператора (`?`) или выражения с использованием `{}` в JSX:

```
class MyReactComponent extends React.Component {
  render() {
    // Не JSX: Использование переменной, if/else или тернарного оператора
    return (
      // JSX: Вывод результата тернарного оператора или выражения в {}
    )
  }
}
```

Мы рассмотрели важные условия для построения интерактивных пользовательских интерфейсов с React и JSX. Время от времени вам придется пояснять функциональность своего красивого, умного кода, чтобы другие люди могли быстрее понять его. Для этого используются комментарии.

3.2.7. Комментарии в JSX

Комментарии в JSX работают по тем же принципам, что и комментарии в обычном JavaScript. Чтобы добавить комментарии в JSX, заключите стандартные комментарии JavaScript в `{}`:

```
let content = (
  <div>
    {/* Как комментарий JS */}
  </div>
)
```

Или же используйте комментарии следующего вида:

```
let content = (
  <div>
    <Post
      /* Я
```

```
      занимаю
      несколько
      строк */
      name={window.isLoggedIn ? window.name : ''} // Внутри JSX
    />
  </div>
)
```

Вы получили некоторое представление о JSX и его преимуществах. Оставшаяся часть этой главы посвящена инструментам JSX и потенциальным ловушкам, которых вы должны избегать.

Прежде чем следовать дальше, вы должны понимать, что для правильного функционирования любого проекта JSX код JSX должен быть *откомпилирован*. Браузеры не могут выполнять JSX — только JavaScript, поэтому вы должны транспилировать код JSX в обычный код JS (см. рис. 3.1).

3.3. Настройка транспилятора JSX с Babel

Как упоминалось ранее, для выполнения кода JSX необходимо преобразовать его в обычный код JavaScript. Этот процесс называется *транспиляцией* (от слов *трансформация* и *компиляция*), и для этой работы существуют различные инструменты. Несколько рекомендованных способов решения этой задачи:

- *Интерфейс командной строки (CLI) Babel* — пакет `babel-cli` предоставляет команду для транспиляции. Этот способ требует меньшей настройки и является самым простым в подготовке.
- *Браузерный сценарий Node.js или JavaScript* — сценарий может импортировать пакет `babel-core` и транспилировать JSX на программном уровне (`babel.transform`). Этот способ открывает возможности для контроля на низком уровне, устраняет абстракции и зависимости от средств сборки и их плагинов.
- *Средства сборки* — такие инструменты, как Grunt, Gulp или Webpack, используют плагин Babel. Это самый популярный способ.

Все эти способы тем или иным образом используют Babel. Babel прежде всего является компилятором ES6+/ES2015+, но он также может преобразовывать JSX в JavaScript. Команда React прекратила разработку собственного преобразователя JSX и рекомендует использовать Babel.

ЕСТЬ ЛИ АЛЬТЕРНАТИВЫ У BABEL 6?

Существуют разные инструменты для транспиляции JSX, но самым популярным — и рекомендованным командой React на официальном сайте React на август 2016 года — является Babel (ранее 5to6). Исторически команда React занималась сопровождением react-tools и JSXTransformer (транспиляция в браузере); но начи-

ная с версии 0.13, команда рекомендует использовать Babel и прервала разработку react-tools и JSXTransformer[1].

Для внутрибраузерной транспиляции на стадии выполнения в Babel версии 5.x существует browser.js — пакет, готовый к использованию. Вы можете подключить его к браузеру (по аналогии с JSXTransformer), и он будет преобразовывать любой код <script> в JS (используйте `type="text/babel"`). Последней версией Babel с browser.js является версия 5.8.34, которую можно подключить прямо из CDN (https://cdnjs.com/libraries/babel-core/5.8.34).

В версии Babel 6.x конфигурации/настройки по умолчанию (такие, как JSX) были отключены, а пакет browser.js удален. Команда Babel предлагает разработчикам создавать собственные пакеты или использовать Babel API. Также существует библиотека babel-standalone (https://github.com/Daniel15/babel-standalone), но ей необходимо указать, какую конфигурацию следует использовать.

Traceur (https://github.com/google/traceur-compiler) — другой инструмент, который может использоваться в качестве замены Babel.

Наконец, TypeScript (www.typescriptlang.org) вроде бы поддерживает компиляцию JSX через jsx-typescript (https://github.com/fdecampredon/jsx-typescript)[2], но это совершенно новый инструментарий и язык (надмножество стандартного JavaScript).

Вероятно, вы сможете использовать JSXTransformer, Babel v5, babel-standalone, TypeScript и Traceur с примерами этой книги (я использую React v15). TypeScript и Traceur — относительно безопасные варианты, потому что они поддерживаются на момент написания книги. Тем не менее, используя что-либо, кроме Babel 6, с примерами этой книги, вы действуете на свой страх и риск. Ни я, ни научные редакторы Manning не тестировали код этой книги, чтобы понять, работает ли он с этими инструментами!

Используя Babel с React, вы можете получить доступ к расширенной функциональности ES6/ES2015, упрощающей процесс разработки, — для этого достаточно добавить дополнительную конфигурацию и модуль для ES6. Шестое поколение стандарта ECMAScript содержит великое множество усовершенствований и на момент написания книги поддерживается всеми *современными* браузерами. Однако у старых браузеров могут возникнуть проблемы с интерпретацией нового кода ES6. Кроме того, если вы хотите использовать ES7, ES8 или ES27, в отдельных браузерах некоторые функции могут быть еще не реализованы.

Babel поможет решить проблему отставания реализации новейших функций ES6 или ES.Next (собирательный термин для большинства новейших функций) в браузерах. Для этого Babel предлагает поддержку следующего поколения языков

[1] Paul O'Shannessy, «Deprecating JSTransform and react-tools,» React, 12 июня 2015 г., http://mng.bz/8yGc.

[2] www.typescriptlang.org/docs/handbook/jsx.html.

JavaScript (многих языков... чувствуете подсказку в названии?). В этом разделе рассматривается рекомендованный способ, использованный в нескольких следующих главах, — Babel CLI, — потому что он обходится минимальной настройкой и не требует знания Babel API (в отличие от способа с API).

Для использования Babel CLI (http://babeljs.io) вам понадобятся Node v6.2.0, npm v3.8.9, babel-cli v6.9.0 (www.npmjs.com/package/babel-cli) и babel-preset-react v6.5.0 (www.npmjs.com/package/babel-preset-react). Работа других версий с кодом из этой книги не гарантирована из-за стремительных изменений в технологиях разработки Node and React.

Если вам нужно установить Node и npm, проще всего загрузить программу установки (общую для Node и npm) с официального сайта: http://nodejs.org. Другие варианты и подробные инструкции по установке Babel приведены в приложении А.

Если вы полагаете, что эти средства уже установлены, или вы не уверены, проверьте версии Node и npm следующими командами оболочки/терминала/командной строки:

```
node -v
npm -v
```

Babel CLI и конфигурация React должны быть доступны локально. Использовать the Babel CLI глобально (-g при установке из npm) не рекомендуется из-за потенциальных конфликтов в тех ситуациях, когда ваши проекты используют разные версии программ. Сокращенная версия инструкций из приложения А:

1. Создайте новую папку (например, ch03/babel-jsx-test).
2. Создайте в новой папке файл package.json и добавьте пустой объект `{}` или же сгенерируйте файл командой `npm init`.
3. Определите свои параметры конфигурации Babel в файле package.json (используется в книге и рассматривается в следующем разделе) или .babelrc (в книге не используется).
4. При желании добавьте в package.json такую информацию, как имя проекта, лицензия, репозиторий GitHub и т. д.
5. Установите Babel CLI и конфигурацию React *локально*, используя команду `npm i babel-cli@6.9.0 babel-preset-react@6.5.0 --save-dev` для сохранения этих зависимостей в объекте `devDependencies` в package.json.
6. При желании создайте сценарий `npm` одной из команд Babel, описанных ниже.

КОНФИГУРАЦИЯ BABEL ES6

Если вам не повезло и приходится обеспечивать поддержку старых браузеров (таких, как IE9), но при этом вы хотите писать код ES6+/ES2015+, потому что это будущий стандарт, вы можете добавить транспилятор babel-preset-es2015 (www.

npmjs.com/package/babel-preset-es2015). Он преобразует ES6 в код ES5. Для этого установите библиотеку:

```
npm i babel-preset-es2015 --save-dev
```

Затем добавьте ее в конфигурацию `presets` вместе с react:

```
{
  "presets": ["react", "es2015"]
}
```

Я не рекомендую использовать транспилятор ES2015, если только вам не приходится поддерживать старые браузеры, по нескольким причинам. Во-первых, вы будете выполнять старый код ES5, который хуже оптимизирован по сравнению с кодом ES6. Во-вторых, вы вводите дополнительную зависимость и повышаете сложность. В-третьих, если уж так много людей продолжают выполнять код ES5 в своих браузерах, то зачем нам — командам поддержки браузеров и обычным разработчикам JavaScript — возиться с ES6? С тем же успехом можно воспользоваться TypeScript (www.typescriptlang.org), ClojureScript (http://clojurescript.org) или CoffeeScript (http://coffeescript.org) — вы получите максимум эффекта за те же деньги!

Повторю то, что написано в приложении А: вам понадобится файл package.json, содержащий как минимум такую настройку:

```
{
  ...
  "babel": {
    "presets": ["react"]
  },
  ...
}
```

Затем выполните следующую команду (из только что созданной папки), чтобы проверить работоспособность версии:

```
$ ./node_modules/.bin/babel --version
```

После установки для преобразования JSX-файла js/script.jsx в js/script.js с кодом JavaScript используется следующая команда:

```
$ ./node_modules/.bin/babel js/script.jsx -o js/script.js
```

Команда получается длинной, потому что она содержит путь к Babel. Эту команду можно сохранить в файле package.json для использования сокращенной версии: `npm run build`. Откройте файл в редакторе и добавьте следующую строку в `scripts`:

```
"build": "./node_modules/.bin/babel js/script.jsx -o js/script.js"
```

Команду можно автоматизировать с помощью ключа `-w` (или `--watch`):

```
$ ./node_modules/.bin/babel js/script.jsx -o js/script.js -w
```

Эта команда Babel отслеживает любые изменения в script.jsx и компилирует их в script.js при сохранении обновленного JSX-файла. Когда это происходит, в терминале/командной строке выводится следующее сообщение:

```
change js/script.jsx
```

Когда у вас накопится больше файлов с кодом JSX, используйте команду с ключом `-d` (`--out-dir`) и именами папок для компиляции исходных файлов JSX (`source`) в файлы JS (`build`):

```
$ ./node_modules/.bin/babel source --d build
```

Часто загрузка одного файла обладает более высокой эффективностью для быстродействия приложения клиентской части, чем загрузка многих файлов, ведь каждый запрос создает дополнительную задержку. Чтобы откомпилировать все файлы в исходном каталоге в один обычный файл JS, используйте ключ `-o` (`--out-file`):

```
$ ./node_modules/.bin/babel src -o script-compiled.js
```

Возможно, в зависимости от конфигурации путей на вашем компьютере вы сможете выполнить команду `babel` вместо `./node_modules/.bin/babel`. В обоих случаях выполнение происходит локально. Если у вас глобально установлена более старая версия `babel-cli`, удалите ее командой `npm rm -g babel-cli`.

Если вам не удается выполнить `babel` при локальной установке `babel-cli` в проекте, попробуйте добавить одну из команд в ваш профиль оболочки: ~/.bash_profile, ~/.bashrc или ~/.zsh в зависимости от оболочки (bash, zsh и т. д.) и поддержки POSIX (UNIX, Linux, macOS и т. д.).

Следующая команда оболочки добавляет путь, чтобы вы могли запускать локально установленные пакеты из npm CLI без его ввода, при наличии файла ./node_modules/.bin в текущем каталоге:

```
if [ -d "$PWD/node_modules/.bin" ]; then
    PATH="$PWD/node_modules/.bin"
fi
```

Сценарий оболочки проверяет, существует ли каталог ./node_modules/.bin в текущей папке терминальной среды bash, и добавляет этот каталог в список путей, чтобы инструменты npm CLI (Babel, Webpack и т. д.) можно было запускать просто по имени: `babel`, `webpack` и т. д.

Еще можно сделать так, чтобы путь был активен постоянно, а не только при наличии подкаталога. Следующая команда оболочки добавляет путь ./node_modules/.bin в переменную среды `PATH` (также в профиле):

```
export PATH="./node_modules/.bin:$PATH"
```

У этого способа есть дополнительное преимущество: он позволит вам локально выполнить *любую программу npm CLI* просто по имени, без обязательного указания пути.

> **СОВЕТ** Чтобы увидеть рабочие примеры конфигураций Babel package.json, откройте проекты из папки ch03 исходного кода, прилагаемого к книге. Они построены по тем же принципам, которые используются в следующих главах. Файл package.json в ch03 содержит сценарии npm build для каждого проекта (подкаталога), нуждающегося в компиляции, если только проект не содержит собственный файл package.json.

Когда вы запускаете сценарий сборки, например `npm run build-hello-world`, он компилирует JSX из ch03/PROJECT_NAME/jsx в обычный код JavaScript и помещает откомпилированный файл в ch03/PROJECT_NAME/js. А следовательно, все, что вам нужно сделать, — это установить необходимые зависимости с использованием `npm i` (при этом создается папка ch03/node_modules), проверить, существует ли сценарий сборки в package.json, а затем выполнить команду `npm run build-PROJECT_NAME`.

То, что описано выше, по моему скромному мнению, является самым простым способом транспиляции JSX в стандартный код JS. Но я хочу, чтобы вы знали о некоторых тонкостях, связанных с React и JSX.

3.4. Потенциальные проблемы React и JSX

В этом разделе рассматриваются некоторые особые случаи. Есть несколько потенциальных проблем, о которых необходимо знать при использовании JSX.

Например, JSX требует наличия слеша (/) в закрывающем теге или, при отсутствии дочерних элементов и использовании одного тега, — в конце этого тега. Например, следующий фрагмент правилен:

```
<a href="http://azat.co">Azat, the master of callbacks</a>
<button label="Save" className="btn" onClick={this.handleSave}/>
```

А этот фрагмент *ошибочен* из-за отсутствия слешей:

```
<a href="http://azat.co">Azat<a>
<button label="Save" className="btn" onClick={this.handleSave}>
```

Формат HTML менее требователен. Многие браузеры игнорируют отсутствующий символ / и прекрасно рендерят элемент без него. Опробуйте `<button>Press me!` и убедитесь сами!

Между HTML и JSX также существуют и другие различия.

3.4.1. Специальные символы

Сущности HTML представляют собой коды для отображения специальных символов, дефисов, кавычек и т. д. Несколько примеров:

```
&copy;
&mdash;
&ldquo;
```

Эти коды можно вывести так же, как любую другую строку, в `<span>` или в строковом атрибуте `<input>`, как и в следующем статическом коде JSX (текст, определяемый в коде без переменных или свойств):

```
<span>&copy;&mdash;&ldquo;</span>
<input value="&copy;&mdash;&ldquo;"/>
```

Но если вы попробуете выводить сущности HTML в `<span>` динамически (из переменной или свойства), то получите непосредственный вывод (`&copy;&mdash;&ldquo;`), а не специальные символы. Таким образом, следующий код работать не будет:

```
// Антипаттерн. НЕ работает!
var specialChars = '&copy;&mdash;&ldquo;'

<span>{specialChars}</span>
<input value={specialChars}/>
```

React/JSX автоматически избегает опасной разметки HTML, что удобно в отношении безопасности (безопасность по умолчанию — классная штука!). Чтобы вывести специальные символы, придется использовать один из следующих способов:

- Разбить их на несколько строк с использованием массива, например `<span>{[<span>&copy;&mdash;&ldquo;</span>]}</span>`. Также можно задать значение `key` (как в `key="specialChars"`), чтобы отключить предупреждение об отсутствующем ключе.
- Скопировать специальный символ прямо в исходный код (проследите за тем, чтобы использовалась кодировка UTF-8).
- Экранировать специальный символ префиксом `\u` и использовать код Юникода (найдите по адресу www.fileformat.info/info/unicode/char/search.htm, если не помните... да и кто их запоминает?).
- Преобразовать код символа в строку вызовом `String.fromCharCode(charCodeNumber)`.
- Использовать внутренний метод `__html` для небезопасного назначения содержимого HTML (http://mng.bz/TplO; не рекомендуется).

Чтобы продемонстрировать последний способ (как крайнее средство — если на «Титанике» все остальное отказало, бегите к шлюпкам!), взгляните на следующий код:

```
var specialChars = {__html: '&copy;&mdash;&ldquo;'}

<span dangerouslySetInnerHTML={specialChars}/>
```

Очевидно, у создателей React есть чувство юмора, раз они выбрали такое название свойства — `dangerouslySetInnerHTML`!

3.4.2. Атрибуты data-

В разделе 2.3 были рассмотрены свойства без учета специфики JSX, но давайте еще раз посмотрим, как создавать нестандартные атрибуты HTML (на этот раз с JSX). В основном React преспокойно игнорирует любые нестандартные атрибуты HTML, добавляемые к компонентам. Неважно, используете вы JSX или традиционный JavaScript, — таково поведение React.

Но иногда бывает нужно передать дополнительные данные с использованием узлов DOM. Такое решение считается антипаттерном, потому что модель DOM не должна использоваться как база данных или локальное хранилище. Но если вы все равно хотите создать нестандартные атрибуты и обработать их, используйте префикс `data-`.

Например, так выглядит действительный нестандартный атрибут `data-object-id`, который React сгенерирует в представлении (разметка HTML будет совпадать со следующим кодом JSX):

```
<li data-object-id="097F4E4F">...</li>
```

Если же на вход подается следующий элемент React/JSX, React не будет генерировать `object-id`, потому что такого стандартного атрибута HTML не существует (в HTML атрибут `object-id` будет отсутствовать, в отличие от следующего кода JSX):

```
<li object-id="097F4E4F">...</li>
```

3.4.3. Атрибут style

Атрибут `style` в JSX работает не так, как в обычной разметке HTML. В JSX вместо строки должен передаваться объект JavaScript, а свойства CSS должны записываться в «верблюжьем регистре». Несколько примеров:

- `background-image` превращается в `backgroundImage`;
- `font-size` превращается в `fontSize`;
- `font-family` превращается в `fontFamily`.

Объект JavaScript можно сохранить в переменной или отрендерить его «на месте» с двойными фигурными скобками (`{{...}}`). Двойные фигурные скобки необходимы, поскольку одна пара нужна для JSX, а другая — для объектного литерала JavaScript.

Предположим, имеется объект с размером шрифта:

```
let smallFontSize = {fontSize: '10pt'}
```

В разметке JSX можно использовать объект `smallFontSize`:

```
<input style={smallFontSize} />
```

Также можно выбрать больший размер шрифта (30 пунктов), передавая значения напрямую без дополнительной переменной:

```
<input style={{fontSize: '30pt'}} />
```

Рассмотрим другой пример прямой передачи стилей. На этот раз `<span>` назначается красная рамка:

```
<span style={{borderColor: 'red',
  borderWidth: 1,
  borderStyle: 'solid'}}>Hey</span>
```

Также сработает следующее значение `border`:

```
<span style={{border: '1px red solid'}}>Hey</span>
```

Главная причина, по которой классы представляются не непрозрачными строками, а объектами JavaScript, это ускорение их обработки React при применении изменений к представлениям.

3.4.4. class и for

React и JSX принимают любые атрибуты, являющиеся стандартными атрибутами HTML, за исключением `class` и `for`. Эти имена являются зарезервированными словами в JavaScript/ECMAScript, а JSX преобразуется в JavaScript. Вместо них следует использовать `className` и `htmlFor`. Например, если вы используете класс `hidden`, его можно определить в `<div>` следующим образом:

```
<div className="hidden">...</div>
```

Если же вам понадобится создать метку для элемента формы, используйте `htmlFor`:

```
<div>
    <input type="radio" name={this.props.name} id={this.props.id}>
    </input>
    <label htmlFor={this.props.id}>
      {this.props.label}
    </label>
</div>
```

3.4.5. Значения логических атрибутов

Наконец, некоторые атрибуты (такие, как `disabled`, `required`, `checked`, `autofocus` и `readOnly`) относятся только к элементам форм. Здесь важнее всего помнить, что значение атрибута должно задаваться в выражении JavaScript (то есть внутри `{}`), а не в строке.

Например, `{false}` используется для разрешения ввода:

```
<input disabled={false} />
```

Но не используйте значение `"false"`, потому что оно пройдет проверку на истинность (непустая строка считается истинным значением в JavaScript — см. врезку), и элемент `input` будет сгенерирован как заблокированный (значение `disabled` интерпретируется как `true`):

```
<input disabled="false" />
```

ИСТИННОСТЬ

В JavaScript/Node квазиистинное (truthy) значение преобразуется в true при обработке в логическом контексте, например в команде if. Значение считается квазиистинным, если оно не является квазиложным (Falsy). (Это официальное определение. Гениально, правда?) Квазиложных значений всего шесть:

- `false`
- `0`
- `""` (пустая строка)
- `null`
- неопределенное
- `NaN` («не число»)

Как видите, строка `"false"` является непустой строкой; следовательно, это квазиистинное значение, которое преобразуется в true. А значит, вы получите `disabled=true` в разметке HTML.

Если опустить значение, то React будет считать, что оно равно `true`:

```
<input disabled />
```

В последующих главах будет использоваться исключительно JSX. Тем не менее очень полезно знать, какой стандартный код JavaScript будет выполняться в браузерах.

3.5. Вопросы

1. Какие из следующих конструкций используются для вывода переменной JavaScript в JSX: `=`, `<%=%>`, `{}` или `<?= ?>`?
2. Атрибут `class` не разрешен в JSX. Да или нет?
3. Для атрибутов, значение которых не задано, по умолчанию используется значение `false`. Да или нет?
4. Встроенный атрибут стиля в JSX представляет собой объект JavaScript, а не строку, в отличие от других атрибутов. Да или нет?

5. Если вы хотите включить логику `if/else` в JSX, вы можете использовать ее в `{}`. Например, `class={if (!this.props.admin) return 'hide'}` — действительный код JSX. Да или нет?

3.6. Итоги

- JSX — всего лишь синтаксический сахар для таких методов React, как `createElement`.
- Вместо стандартных атрибутов HTML `class` и `for` следует использовать `className` и `htmlFor`.
- Атрибут `style` получает объект JavaScript, а не строку, как с обычными атрибутами HTML.
- Тернарные операторы и IIFE — лучшие способы реализации команд `if/else`.
- Задачи вывода переменных, написания комментариев и сущностей HTML, а также компиляции кода JSX в стандартный код JavaScript решаются достаточно легко.
- Есть несколько способов преобразования JSX в обычный код JavaScript: компиляция с Babel CLI требует минимальной настройки по сравнению с настройкой процесса сборки с такими инструментами, как Gulp или Webpack, или написанием сценариев Node/JavaScript для использования Babel API.

3.7. Ответы

1. Для переменных и выражений используется `{}`.
2. Да. `class` — зарезервированное слово или специальная команда JavaScript. По этим причинам в JSX используется `className`.
3. Нет. Для явного задания логических значений рекомендуется использовать синтаксис `attribute_name={false/true}`.
4. Да. `style` является объектом по соображениям быстродействия.
5. Нет. Во-первых, `class` не является полноценным атрибутом. Во-вторых, вместо `if return` следует использовать тернарный оператор.

4 Состояния и их роль в интерактивной природе React

Посмотрите вступительный видеоролик к этой главе, отсканировав QR-код или перейдя по ссылке http://reactquickly.co/videos/ch04.

ЭТА ГЛАВА ОХВАТЫВАЕТ СЛЕДУЮЩИЕ ТЕМЫ:

- Состояние компонента React.
- Работа с состояниями.
- Состояния и свойства.
- Компоненты с состоянием и без состояния.

Если бы вам пришлось прочитать в этой книге всего одну главу — стоило бы выбрать именно эту! Без состояний компоненты React остаются не более чем усовершенствованными статическими шаблонами. Надеюсь, вы разделяете мой энтузиазм, потому что понимание концепций этой главы позволит вам строить намного более интересные приложения.

Представьте, что вы строите поле ввода с автозаполнением (рис. 4.1). При вводе данных поле должно выдать запрос к серверу, чтобы получить информацию о подходящих вариантах для отображения вывода на веб-странице. До сих пор вы работали со свойствами, и знаете, что изменение свойств позволяет получить разные представления. Однако свойства не могут изменяться в контексте текущего компонента, потому что они передаются при создании компонента.

Иначе говоря, свойства неизменяемы в текущем компоненте, а это означает, что вы не можете изменять свойства в этом компоненте, если только не создадите

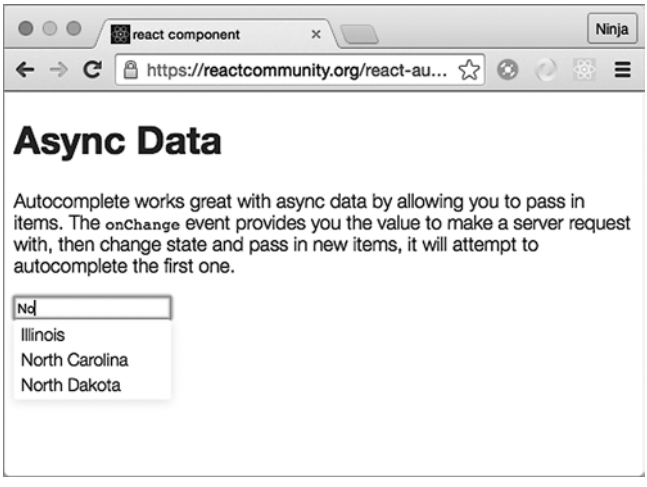


Рис. 4.1. Компонент react-autocomplete в действии

компонент заново и не передадите новые значения от родителя (рис. 4.2). Но информацию, полученную от сервера, нужно где-то сохранить, а затем вывести новый список вариантов в представлении. Как обновить представление, если свойства не могут изменяться?

Одно из возможных решений — рендерить элемент с новыми свойствами каждый раз, когда вы получаете новый ответ от сервера. Но тогда вам придется разместить

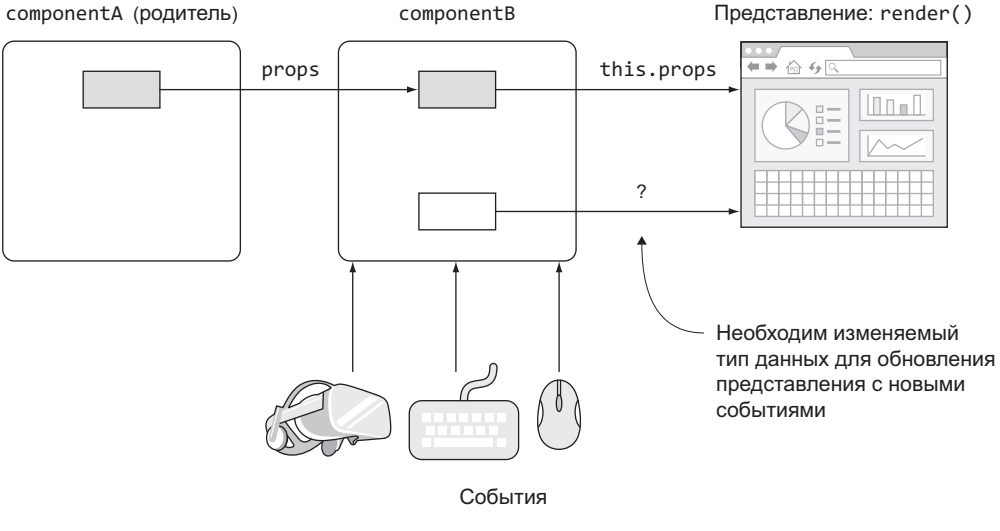


Рис. 4.2. Для изменения представления необходим другой тип данных, который может изменяться в компоненте

логику за пределами компонента — и компонент перестает быть самодостаточным. Очевидно, если значения свойств нельзя изменять, а автозаполнение должно быть самодостаточным, использовать свойства невозможно. Тогда возникает вопрос: как обновлять представления в ответ на события без повторного создания компонента (`createElement()` или `JSX <NAME/>`)? Именно эту проблему решают *состояния*.

После того как ответ от сервера будет готов, код обратного вызова изменит состояние компонента соответствующим образом. Вам придется написать этот код самостоятельно. Однако после того как состояние будет обновлено, React автоматически обновит представление за вас (только в тех местах, где оно должно быть обновлено, то есть там, где используются данные состояния).

С состоянием компонентов React вы можете строить интерактивные, содержательные приложения React. *Состояние* — основополагающая концепция, которая позволяет строить компоненты React, способные хранить данные и автоматически обновлять представления в соответствии с изменениями в данных.

ПРИМЕЧАНИЕ Исходный код примеров этой главы доступен по адресам www.manning.com/books/react-quickly и https://github.com/azat-co/react-quickly/tree/master/ch04 (в папке ch04 репозитория GitHub https://github.com/azat-co/react-quickly). Также некоторые демонстрационные примеры доступны по адресу http://reactquickly.co/demos.

4.1. Что такое состояние компонента React?

Состояние React представляет собой изменяемое хранилище данных компонента — автономные функционально-ориентированные блоки пользовательского интерфейса и логики. «Изменяемость» означает, что значения состояний могут изменяться. Используя состояние в представлении (`render()`) и изменяя значения позднее, вы можете влиять на внешний вид представления.

Метафора: если представить себе компонент в виде функции, на вход которой передаются свойства и состояние, то результатом функции будет описание пользовательского интерфейса (представление). Свойства и состояния расширяют представления, но они используются для разных целей (см. раздел 4.3).

Работая с состояниями, вы обращаетесь к ним по имени. Имя является атрибутом (то есть ключом объекта или свойством объекта — не свойством компонента) объекта `this.state`, например `this.state.autocompleMatches` или `this.state.inputFieldValue`.

ПРИМЕЧАНИЕ Вообще, термин «состояния» относится к атрибутам объекта `this.state` в компоненте. В зависимости от контекста, состояние (в единственном числе) может относиться к объекту `this.state` или отдельному атрибуту (такому, как `this.state.inputFieldValue`). С другой стороны, термин «состояния» (во множественном числе) почти всегда относится к множественным атрибутам объекта state в одном компоненте.

Данные состояния часто используются для отображения динамической информации в представлении для расширения рендера представлений. Возвращаясь к более раннему примеру поля с автозаполнением: состояние изменяется в ответ на запрос XHR к серверу, который, в свою очередь, инициируется вводом данных в поле. React обеспечивает актуализацию представлений при изменении состояния, используемого в представлениях. На деле при изменении состояния изменяются только соответствующие части представлений (до отдельных элементов и даже значений атрибутов отдельного элемента).

Все остальное в DOM остается неизменным. Это возможно благодаря виртуальной модели DOM (см. раздел 1.1.1), которую React использует для определения дельты (совокупности изменений) в процессе согласования. Именно этот факт позволяет писать код в декларативном стиле. React выполняет за вас всю рутинную работу. Основные этапы изменения представления рассматриваются в главе 5.

Разработчики React используют состояния для генерирования новых пользовательских интерфейсов. Свойства компонентов (`this.props`), обычные переменные (`inputValue`) и атрибуты классов (`this.inputValue`) для этого не подойдут, потому что изменение их значений (в контексте текущего компонента) не инициирует изменения представления. Например, следующий фрагмент является антипаттерном, который показывает, что изменение значения в любом месте, кроме состояния, не приведет к обновлению представления:

```
// Антипаттерн: не делайте так!
let inputValue = 'Texas'
class Autocomplete extends React.Component {
  updateValues() {  ◄—— Инициируется в результате действия пользователя (ввод данных)
    this.props.inputValue = 'California'
    inputValue = 'California'
    this.inputValue = 'California'
  }
  render() {
    return (
      <div>
        {this.props.inputValue}
        {inputValue}
        {this.inputValue}
      </div>
    )
  }
}
```

А теперь посмотрим, как работать с состояниями компонентов React.

> **ПРИМЕЧАНИЕ** Как упоминалось ранее (повторение — мать учения), свойства будут изменять представление при передаче нового значения от родителя, что, в свою очередь, приведет к созданию нового экземпляра компонента, с которым вы работаете. В контексте конкретного компонента простое изменение свойств, как в примере `this.props.inputValue = 'California'`, ничего не даст.

4.2. Работа с состояниями

Чтобы работать с состояниями, вы должны уметь обращаться к значениям, обновлять их и задавать исходные значения. Начнем с обращения к состояниям в компонентах React.

4.2.1. Обращение к состояниям

Объект `state` является атрибутом компонента, а обращаться к нему следует через ссылку `this`, например `this.state.name`. Как вы помните, к переменным можно обращаться и выводить их в коде JSX в фигурных скобках (`{}`). Аналогичным образом в `render()` можно выполнить рендер `this.state` (как и любую другую переменную или атрибут класса нестандартного компонента), например `{this.state.inputFieldValue}`. Этот синтаксис аналогичен синтаксису обращения к свойствам в `this.props.name`.

Используем то, что вы узнали, для реализации часов на рис. 4.3. Наша цель — создать автономный класс компонента, который любой желающий сможет импортировать и использовать в своем приложении без особых хлопот. На часах должно отображаться текущее время.

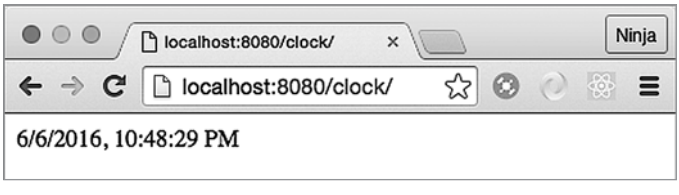


Рис. 4.3. Компонент часов выводит текущее время в цифровом формате. Показания часов обновляются каждую секунду

Проект имеет следующую структуру:

```
/clock
  index.html
  /jsx
    script.jsx
    clock.jsx
  /js
    script.js
    clock.js
    react.js
    react-dom.js
```

Я использую Babel CLI с флагами отслеживания (`-w`) и каталога (`-d`) для компиляции всех исходных файлов JSX из clock/jsx в целевую папку clock/js и перекомпиляции при обнаружении изменений. Кроме того, я сохранил команду как сценарий npm в файле package.json родительской папки ch04 для выполнения команды `npm run build-clock` из ch04:

```
"scripts": {
  "build-clock": "./node_modules/.bin/babel clock/jsx -d clock/js -w"
},
```

Разумеется, время не стоит на месте (нравится нам это или нет). Из-за этого необходимо постоянно обновлять представление, а для этого можно воспользоваться состоянием. Присвойте ему имя `currentTime` и попробуйте организовать рендер состояния так, как показано в листинге 4.1.

Листинг 4.1. Рендер состояния в JSX

```
class Clock extends React.Component {
  render() {
    return <div>{this.state.currentTime}</div>
  }
}

ReactDOM.render(
  <Clock />,
  document.getElementById('content')
)
```

Вы получите сообщение об ошибке: `Uncaught TypeError: Cannot read property 'currentTime' of null`. Обычно от сообщений об ошибках JavaScript пользы примерно столько же, сколько от стакана холодной воды для утопающего. Хорошо, что по крайней мере в этом случае JavaScript выводит осмысленное сообщение.

Из сообщения следует, что значение `currentTime` не определено. В отличие от свойств, состояния не задаются в родителе. Вызвать `setState` в `render()` тоже не получится, потому что это создаст цикл (`setState` ▸ `render` ▸ `setState`...), — и React сообщит об ошибке.

4.2.2. Назначение исходного состояния

Вы уже видели, что перед использованием данных состояния в `render()` необходимо инициализировать состояние. Чтобы задать исходное состояние, используйте `this.state` в конструкторе с синтаксисом класса ES6 `React.Component`. Не забудьте вызвать `super()` со свойствами; в противном случае логика в родителе (`React.Component`) не сработает:

```
class MyFancyComponent extends React.Component {
  constructor(props) {
    super(props)
    this.state = {...}
  }
  render() {
    ...
  }
}
```

При назначении исходного состояния также можно добавить другую логику — например, задать значение `currentTime` с использованием `new Date()`. Вы даже можете использовать `toLocaleString()` для получения правильного формата даты/времени для текущего местонахождения пользователя, как показано ниже (ch04/clock).

Листинг 4.2. Конструктор компонента Clock

```
class Clock extends React.Component {
  constructor(props) {
    super(props)
    this.state = {currentTime: (new Date()).toLocaleString()}
  }
  ...
}
```

Значение `this.state` должно быть объектом. Мы не будем углубляться в подробности `constructor()` из ES6; обращайтесь к приложению Д и сводке ES6 по адресу https://github.com/azat-co/cheatsheets/tree/master/es6. Суть в том, что, как и в других ООП-языках, конструктор (то есть `constructor()`) вызывается при создании экземпляра класса. Имя метода-конструктора должно быть именно таким; считайте это одним из правил ES6. Кроме того, при создании метода `constructor()` в него почти всегда должен включаться вызов `super()`, без которого конструктор родителя не будет выполнен. С другой стороны, если вы не определите метод `constructor()`, то вызов `super()` будет предполагаться по умолчанию.

АТРИБУТЫ КЛАССА

Будем надеяться, что TC39 (Technical Committee 39: люди, стоящие за стандартом ECMAScript) добавит атрибуты в синтаксис классов в будущих версиях ECMAScript! Тогда вы сможете задавать состояние не только в конструкторе, но и в теле класса:

```
class Clock extends React.Component {
  state = {
    ...
  }
}
```

С предложением относительно полей/атрибутов/свойств классов можно ознакомиться по адресу https://github.com/jeffmo/es-class-fields-and-static-properties. Оно существует уже много лет, но на момент написания книги (март 2017 г.) продолжает находиться на стадии 2 (стадия 4 означает принятие и включение в стандарт); это означает, что оно не имеет широкой поддержки в браузерах. А значит, данная возможность не будет работать по умолчанию. (На момент написания книги не было ни одного браузера с поддержкой полей классов.) Скорее всего, вам придется воспользоваться транспилятором (Babel, Traceur или TypeScript), чтобы код работал во всех браузерах. За текущей информацией о совместимости или свойствах классов можно обратиться к таблице совместимости ECMAScript (http://kangax.github.io/compat-table/esnext), а при необходимости воспользоваться конфигурацией Babel ES.Next.

Имя `currentTime` выбрано произвольно; вы должны использовать это же имя позднее, при чтении и обновлении этого состояния.

Объект `state` может содержать вложенные объекты или массивы. В следующем примере в состояние добавляется массив с описаниями книг:

```
class Content extends React.Component {
  constructor(props) {
    super(props)
    this.state = {
      githubName: 'azat-co',
      books: [
        'pro express.js',
        'practical node.js',
        'rapid prototyping with js'
      ]
    }
  }
  render() {
    ...
  }
}
```

Метод `constructor()` вызывается всего один раз, при создании элемента React на базе класса. Таким образом, задать состояние напрямую с использованием `this.state` можно только один раз — в методе `constructor()`. Не устанавливайте и не обновляйте состояние напрямую с помощью `this.state = ...` где-то еще, так как это может привести к непредвиденным последствиям.

> **ПРИМЕЧАНИЕ** При определении компонентов с использованием метода React `createClass()` необходимо использовать метод `getInitialState()`. За дополнительной информацией о `createClass` и примером для ES5 обращайтесь к врезке в разделе 2.2, «ES6+/ES2015+ и React».

Так вы получите только исходное значение, которое очень быстро устареет — всего за 1 секунду. Кому нужны часы, которые не показывают текущее время? К счастью, существует механизм обновления текущего состояния.

4.2.3. Обновление состояния

Состояние изменяется методом класса `this.setState(data, callback)`. При вызове этого метода React объединяет данные с текущими состояниями и вызывает `render()`, после чего вызывает `callback`.

Определение обратного вызова `callback` в `setState()` важно, потому что метод работает *асинхронно*. Если работа приложения зависит от нового состояния, вы можете воспользоваться этим обратным вызовом, чтобы убедиться в том, что новое состояние стало доступным.

Если вы просто будете считать, что состояние обновилось, не дожидаясь завершения `setState()`, то есть работать синхронно при выполнении асинхронной операции, может возникнуть ошибка: работа программы зависит от обновления значений состояния, а состояние остается старым.

До сих пор мы рендерили время из состояния. Вы уже знаете, как задать исходное состояние, но ведь оно должно обновляться каждую секунду, верно? Для этого нужно использовать функцию-таймер браузера `setInterval()` (http://mng.bz/P2d6), которая будет проводить обновление состояния каждые `n` миллисекунд. Метод `setInterval()` реализован практически во всех современных браузерах как глобальный, а это означает, что он может использоваться без каких-либо дополнительных библиотек или префиксов. Пример:

```
setInterval(()=>{
  console.log('Updating time...')
  this.setState({
    currentTime: (new Date()).toLocaleString()
  })
}, 1000)
```

Чтобы запустить отсчет времени, необходимо вызвать `setInterval()` всего один раз. Создадим метод `launchClock()` исключительно для этой цели; `launchClock()` будет вызываться в конструкторе. Итоговая версия компонента приведена в листинге 4.3 (ch04/clock/jsx/clock.jsx).

Листинг 4.3. Реализация часов с состоянием

```
class Clock extends React.Component {
            constructor(props) {
              super(props)
              this.launchClock()
              this.state = {
                currentTime: (new Date()).toLocaleString()
              }
            }
            launchClock() {
              setInterval(()=>{
                console.log('Updating time...')
                this.setState({
                  currentTime: (new Date()).toLocaleString()
                      })
              }, 1000)
          }
          render() {
            console.log('Rendering Clock...')
            return <div>{this.state.currentTime}</div>
          }
        }
```

Запускает launchClock()

Приводит исходное состояние к текущей дате

Обновляет состояние текущим временем каждую секунду

Рендерит состояние

Метод `setState()` может вызываться где угодно, не только в методе `launchClock()` (который вызывается в конструкторе), как в примере. Обычно метод `setState()` вызывается из обработчика событий или в качестве обратного вызова при поступлении или обновлении данных.

> **СОВЕТ** Попытка изменения состояния в коде командой вида `this.state.name= 'new name'` ни к чему не приведет. Она не приведет к повторному рендеру и обновлению реальной модели DOM, чего бы вам хотелось. В большинстве случаев прямое изменение состояния без `setState()` является антипаттерном, и его следует избегать.

Важно заметить, что метод `setState()` обновляет только те состояния, которые ему были переданы (частично или со слиянием, но без полной замены). Он не заменяет весь объект `state` каждый раз. Следовательно, если изменилось только одно из трех состояний, два других останутся неизменными. В следующем примере `userEmail` и `userId` изменяться не будут:

```
constructor(props) {
  super(props)
  this.state = {
    userName: 'Azat Mardan',
    userEmail: 'hi@azat.co',
    userId: 3967
  }
}
updateValues() {
  this.setState({userName: 'Azat'})
}
```

Если вы намерены обновить все три состояния, это придется сделать явно, передав новые значения этих состояний `setState()`. (Также в старом коде, который сейчас уже не работает, иногда встречается метод `this.replaceState()`; он официально признан устаревшим[1]. Как нетрудно догадаться по имени, он заменял весь объект `state` со всеми его атрибутами.)

Помните, что вызов `setState()` инициирует выполнение `render()`. В большинстве случаев он работает. В некоторых особых случаях, в которых код зависит от внешних данных, можно инициировать повторный рендер вызовом `this.forceUpdate()`. Тем не менее такие решения нежелательны, потому что опора на внешние данные (вместо состояния) делает компоненты менее надежными и зависящими от внешних факторов (жесткое связывание).

Как упоминалось ранее, к объекту `state` можно обращаться в записи `this.state`. В JSX выводимые значения заключаются в фигурные скобки (`{}`), следовательно, для объявления свойства состояния в представлении (то есть в команде `return` метода `render`) следует применить запись `{this.state.NAME}`.

---

[1] https://github.com/facebook/react/issues/3236.

Волшебство React наступает тогда, когда вы используете данные состояния в представлении (например, при выводе, в команде `if/else`, как значение атрибута или значение свойства дочернего элемента), а затем передаете `setState()` новые значения. Бах! React обновляет всю необходимую разметку HTML за вас. В этом можно убедиться в консоли DevTools, где должны отображаться циклы «Updating...» и «Rendering...». А самое замечательное, что это повлияет *только* на абсолютный минимум необходимых элементов DOM.

СВЯЗЫВАНИЕ THIS В JAVASCRIPT

В JavaScript `this` изменяет свое значение в зависимости от места, из которого вызывается функция. Чтобы убедиться в том, что this относится к вашему классу компонента, необходимо связать функцию в правильном контексте (значение this: ваш класс компонента).

Если вы используете ES6+/ES2015+, как я в этой книге, используйте синтаксис `=>` для создания функции с автоматическим связыванием:

```
setInterval(()=>{
  this.setState({
    currentTime: (new Date()).toLocaleString()
  })
}, 1000)
```

Под автоматическим связыванием (autobinding) имеется в виду то, что функция, созданная с `=>`, получает текущее значение `this` — в данном случае `Clock`.

Также существует ручное решение, применяющее метод `bind(this)` к замыканию:

```
function() {...}.bind(this)
```

Для компонента часов это выглядит так:

```
setInterval(function(){
  this.setState({
    currentTime: (new Date()).toLocaleString()
  })
}.bind(this), 1000)
```

Это поведение не является специфической особенностью React. Ключевое слово `this` изменяется в замыкании функции, поэтому какая-то разновидность связывания необходима; вы также можете сохранить контекст (значение `this`) для использования в будущем.

Обычно для сохранения исходного значения `this` используются такие переменные, как `self`, `that` и `_this`. Вероятно, вы видели команды следующего вида:

```
var that = this
var _this = this
var self = this
```

Идея проста: вы создаете переменную и используете ее в замыкании, вместо того чтобы обращаться к `this`. Новая переменная является не копией, а всего лишь ссылкой на исходное значение `this`. Вот как выглядит функция `setInterval()`:

```
var _this = this
setInterval(function(){
  _this.setState({
    currentTime: (new Date()).toLocaleString()
  })
}, 1000)
```

Часы успешно работают, как показано на рис. 4.4. Тадам!

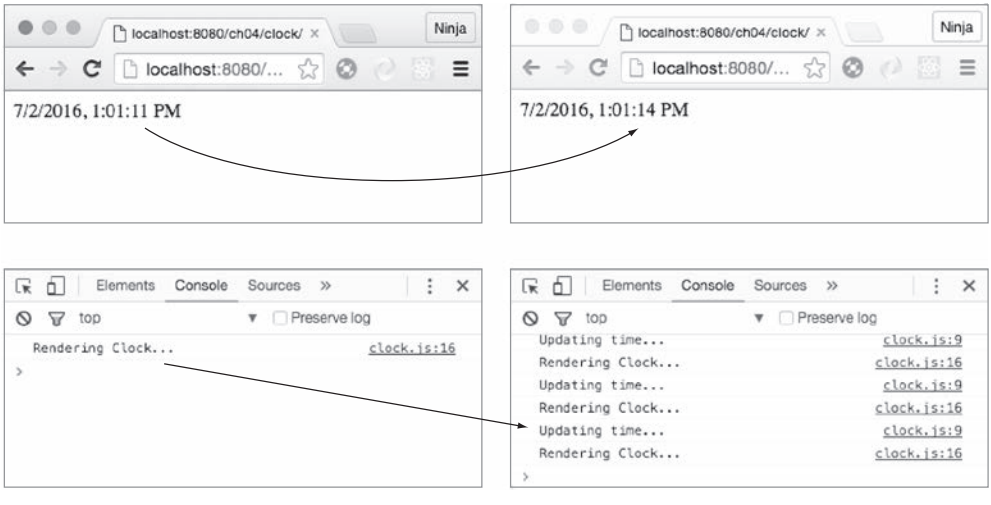


Рис. 4.4. Компонент Clock отсчитывает время

И последнее замечание перед тем, как двигаться дальше. Вы можете проследить за тем, как React повторно использует тот же элемент DOM `<div>` и изменяет только содержащийся в нем текст. Воспользуйтесь DevTools для изменения CSS этого элемента. Я добавил стиль `color: blue`, чтобы текст выводился синим цветом (рис. 4.5). Я создал встроенный стиль, а не класс. Элемент и его новый встроенный стиль оставались неизменными, пока отсчет времени продолжался.

React обновляет только внутреннюю разметку HTML (содержимое второго контейнера `<div>`). Сам элемент `<div>`, как и другие элементы на этой странице, остаются неизменными. Удобно.

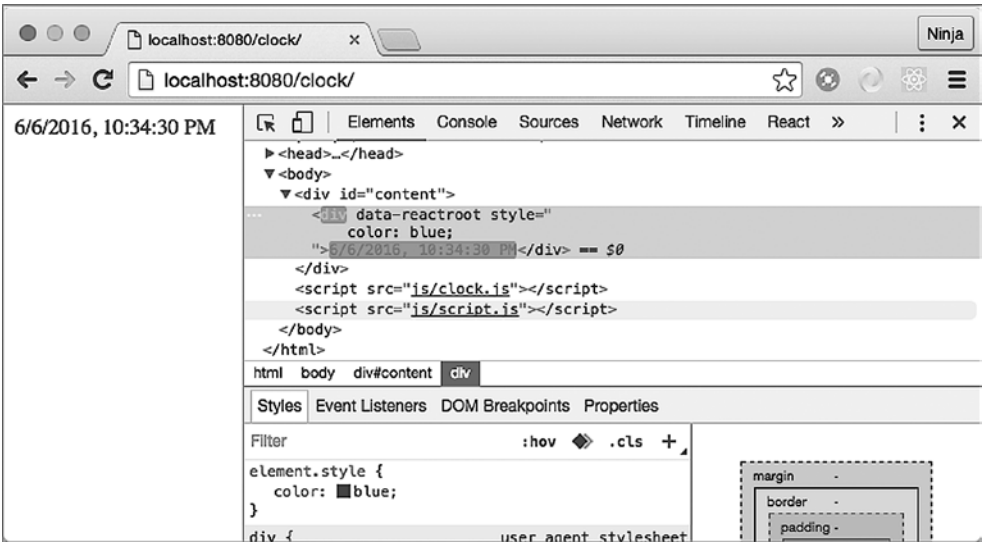


Рис. 4.5. React обновляет текст времени, а не элемент (я вручную добавил стиль color: blue, и элемент остался синим)

4.3. Состояния и свойства

Состояния и свойства являются атрибутами класса, то есть для обращения к ним используется запись `this.state` и `this.props`. И это единственное сходство! Одно из главных различий между состояниями и свойствами заключается в том, что первые могут изменяться, а вторые неизменяемы.

Другое различие между свойствами и состояниями заключается в том, что свойства передаются из родительских компонентов, тогда как состояния определяются в самом компоненте, а не в его родителе. Как следствие, изменить значение свойства можно только из родителя, а не из компонента. Итак, свойства определяют представление при создании, после чего остаются статическими (не изменяются). С другой стороны, состояние задается и обновляется объектом.

Свойства и состояния служат разным целям, однако и те и другие доступны в виде атрибутов класса компонента и помогают составлять компоненты с разным представлением. Различия между свойствами и состояниями проявляются в том, что касается жизненного цикла компонента (подробнее см. в главе 5). Свойства и состояния можно рассматривать как входные данные для функции, которая выдает разные результаты. Эти результаты являются представлениями. Таким образом, вы можете определить разные пользовательские интерфейсы (представления) для каждого набора разных свойств и состояний (рис. 4.6).

Не все компоненты должны обладать состоянием. А в следующем разделе я покажу, как использовать свойства с компонентами без состояния.

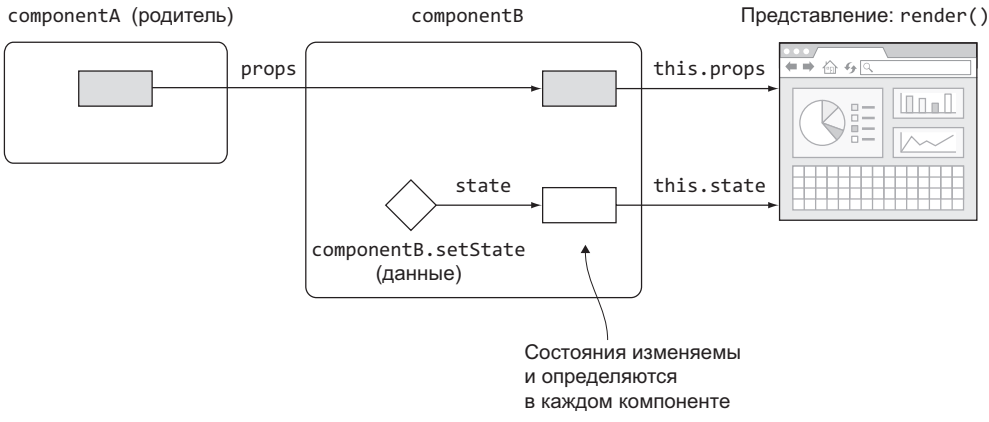


Рис. 4.6. Новые значения для свойств и состояний могут изменить пользовательский интерфейс. Новые значения свойств поступают от родителя, а новые значения состояния — от компонента

4.4. Компоненты без состояния

Компонент без состояния (stateless) не содержит состояний, других компонентов или других событий/методов жизненного цикла React (см. главу 5). Цель компонентов без состояния — просто рендеринг представления. Единственное, что может делать компонент без состояния, — получать свойства и делать что-то с ними; по сути, это простая функция с входными данными (свойствами) и выводом (UI-элемент).

Преимущество использования компонентов без состояния — полная предсказуемость, потому что у них есть один набор входных данных, который определяет результат. Предсказуемость означает, что такие элементы проще в понимании, сопровождении и отладке. Собственно, отсутствие состояния является самой желанной практикой React: чем больше компонентов без состояния вы используете и чем меньше компонентов с состоянием, тем лучше.

В первых трех главах вы написали множество компонентов без состояния. Например, программа Hello World является компонентом без состояния (ch03/hello-js-world-jsx/jsx/script.jsx).

Листинг 4.4. Hello World без состояния

```
class HelloWorld extends React.Component {
  render() {
    return <h1 {...this.props}>Hello {this.props.frameworkName} world!!!
      ➥ </h1>
  }
}
```

Чтобы у компонентов без состояния был более компактный синтаксис, React использует функциональный стиль: вы создаете функцию, которая получает свойства в аргументе и возвращает представление. Компонент без состояния рендерится, как любой другой компонент. Например, компонент `HelloWorld` можно переписать в виде функции, возвращающей `<h1>`:

```
const HelloWorld = function(props){
  return <h1 {...props}>Hello {props.frameworkName} world!!!</h1>
}
```

Также для компонентов без состояния можно использовать «стрелочные» функции ES6+/ES2015+. Следующий фрагмент аналогичен предыдущему (`return` тоже можно опустить, но я предпочитаю включать эту команду):

```
const HelloWorld = (props)=>{
  return <h1 {...props}>Hello {props.frameworkName} world!!!</h1>
}
```

Таким образом, вы также можете определять функции как компоненты React без необходимости в состоянии. Иначе говоря, чтобы создать компонент без состояния, определите его в виде функции. В следующем примере `Link` является компонентом без состояния:

```
function Link (props) {
  return <a href={props.href} target="_blank" className="btn btn-primary">
  ➥ {props.text}</a>
}
ReactDOM.render(
  <Link text='Buy React Quickly'
  ➥ href='https://www.manning.com/books/react-quickly'/>,
  document.getElementById('content')
)
```

И хотя в автоматическом связывании нет необходимости, вы можете использовать синтаксис «стрелочных» функций для краткости (если команда только одна, запись может занимать всего одну строку):

```
const Link = props => <a href={props.href}
  target="_blank"
  className="btn btn-primary">
    {props.text}
</a>
```

Или вы можете использовать ту же «стрелочную» функцию с фигурными скобками (`{}`), явным возвратом и круглыми скобками (`()`), чтобы она *субъективно* лучше читалась:

```
const Link = (props)=> {
  return (
    <a href={props.href}
```

```
      target="_blank"
      className="btn btn-primary">
        {props.text}
    </a>
  )
}
```

Компонент без состояния не может обладать состоянием, но может содержать два свойства: `propTypes` и `defaultProps` (см. разделы 8.1 и 8.2 соответственно). Эти свойства задаются для объекта. И кстати говоря, ставить открывающую круглую скобку после `return` *не* обязательно — при условии, что элемент начинается в той же строке:

```
function Link (props) {
  return <a href={props.href}
    target="_blank"
    className="btn btn-primary">
      {props.text}
  </a>
}
Link.propTypes = {...}
Link.defaultProps = {...}
```

Также *нельзя* использовать ссылки (`refs`) с компонентами по умолчанию (функциями[1]). Если же без использования `refs` не обойтись, компонент без состояния можно упаковать в обычный компонент React. О ссылках подробнее рассказано в разделе 7.2.3.

4.5. Компоненты с состоянием и без состояния

Почему стоит использовать компоненты без состояния? Они более декларативны и лучше работают тогда, когда все, что вам нужно, — отрендерить разметку HTML без создания экземпляра или компонентов жизненного цикла. По сути, компоненты без состояния сокращают дублирование, предоставляют улучшенный синтаксис и упрощают код, когда ваши потребности ограничиваются смешением нескольких свойств и элементов в HTML.

Лично я рекомендую (и команда React считает это правильным подходом) использовать компоненты без состояния вместо обычных компонентов настолько часто, насколько возможно. Но как вы видели в примере с часами, это возможно не всегда; иногда приходится прибегать к использованию состояний. Итак, на вершине иерархии располагаются компоненты с состоянием, которые обрабатывают состояния пользовательского интерфейса, взаимодействия и другую логику приложения (например, загрузку данных с сервера).

---

[1] «React stateless component this.refs..value?», http://mng.bz/Eb91.

Не стоит полагать, что компоненты без состояния должны быть статическими. Передавая им разные свойства, можно изменять их представление. А теперь рассмотрим пример, в котором проведем рефакторинг и расширение `Clock` до трех компонентов: компонента часов с состоянием и логикой его обновления и двух компонентов без состояния `DigitalDisplay` и `AnalogDisplay`, которые только выводят время (но делают это по-разному). Нужно реализовать нечто вроде показанного на рис. 4.7. Красиво, правда?

Проект имеет следующую структуру:

```
/clock-analog-digital
  /jsx
    analog-display.jsx
    clock.jsx
    digital-display.jsx
    script.jsx
  /js
    analog-display.js
    clock.js
    digital-display.js
    script.js
    react.js
    react-dom.js
  index.html
```

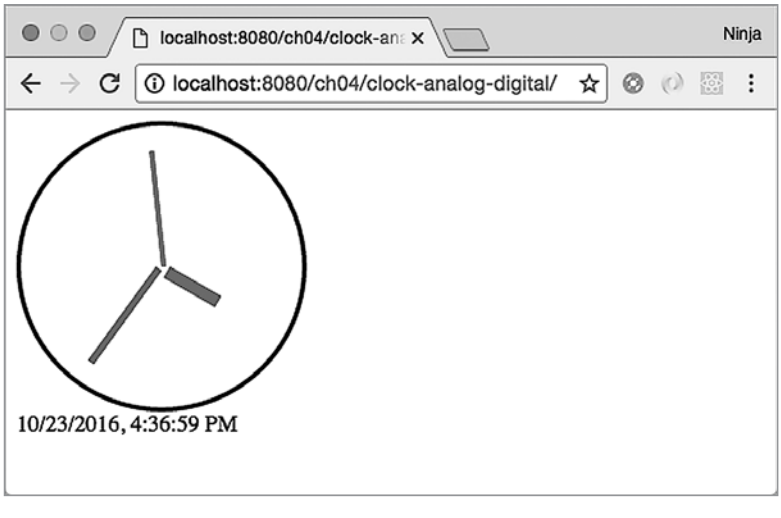


Рис. 4.7. Часы с двумя режимами вывода: цифровым и аналоговым

Код `Clock` ренедерит два дочерних элемента и передает свойство `time` со значением состояния `currentTime`. Состояние родителя становится свойством дочернего элемента.

Листинг 4.5. Передача состояния дочерним элементам

```
...
  render() {
    console.log('Rendering...')
    return <div>
      <AnalogDisplay time={this.state.currentTime}/>
      <DigitalDisplay time={this.state.currentTime}/>
    </div>
  }
```

Теперь нужно создать компонент `DigitalDisplay`; сделать это несложно. Нужно написать функцию, которая получает объект свойств и выводит `time` из этого аргумента (`props.time`), как показано далее (ch04/clock-analog-digital/jsx/digital-display.jsx).

Листинг 4.6. Компонент цифрового отображения времени без состояния

```
const DigitalDisplay = function(props) {
  return <div>{props.time}</div>
}
```

`AnalogDisplay` также представляет собой функцию, реализующую компонент без состояния; но в теле этой функции программируется анимация поворота стрелок. В анимации используется свойство `time`, ни от какого состояния она не зависит. Время передается в виде строки; оно преобразуется в объект `Date`, после чего из него выделяются часы, минуты и секунды, которые затем преобразуются в градусы. Например, преобразование секунд в градусы выполняется так:

```
let date = new Date('1/9/2007, 9:46:15 AM')
console.log((date.getSeconds()/60)*360 ) // 90
```

Зная угол в градусах, можно использовать их в коде CSS, записанном в виде объектного литерала. При этом в коде CSS React стилевые свойства записываются в «верблюжьем регистре», тогда как в обычном коде CSS из-за дефисов (`-`) стилевые свойства становятся недействительным кодом JavaScript. Как упоминалось ранее, использование объектов для стилей позволяет React быстрее определять различия между старым и новым элементом. За дополнительной информацией о `style` и CSS в React обращайтесь к разделу 3.4.3.

В листинге 4.7 приведен компонент аналогового отображения времени без состояния с кодом CSS, использующим значения из свойства `time` (ch04/clock-analog-digital/jsx/analog-display.jsx).

Листинг 4.7. Компонент аналогового отображения времени без состояния

```
const AnalogDisplay = function AnalogDisplay(props) {
  let date = new Date(props.time)
  let dialStyle = {
    position: 'relative',
    top: 0,
    left: 0,
```

Преобразует строку с датой в объект для последующего разбора

```
    width: 200,
    height: 200,
    borderRadius: 20000,
    borderStyle: 'solid',
    borderColor: 'black'
  }
  let secondHandStyle = {
    position: 'relative',
    top: 100,
    left: 100,
    border: '1px solid red',
    width: '40%',
    height: 1,
    transform: 'rotate(' + ((date.getSeconds()/60)*360 - 90 )
    ➥ .toString() + 'deg)',
    transformOrigin: '0% 0%',
    backgroundColor: 'red'
  }
  let minuteHandStyle = {
    position: 'relative',
    top: 100,
    left: 100,
    border: '1px solid grey',
    width: '40%',
    height: 3,
    transform: 'rotate(' + ((date.getMinutes()/60)*360 - 90 )
    ➥ .toString() + 'deg)',
    transformOrigin: '0% 0%',
    backgroundColor: 'grey'
  }
  let hourHandStyle = {
    position: 'relative',
    top: 92,
    left: 106,
    border: '1px solid grey',
    width: '20%',
    height: 7,
    transform: 'rotate(' + ((date.getHours()/12)*360 - 90 ).toString() +
               'deg)',
    transformOrigin: '0% 0%',
    backgroundColor: 'grey'
  }
  return <div>
    <div style={dialStyle}>
      <div style={secondHandStyle}/>
      <div style={minuteHandStyle}/>
      <div style={hourHandStyle}/>
    </div>
  </div>
}
```

Использует borderRadius (border-radius в обычном коде CSS) с с большим значением относительно ширины, чтобы создать круг

Использует transformOrigin для смещения центра поворота

Вычисляет угол и поворачивает вторую стрелку со смещением, уменьшенным на 90, чтобы установить местоположение начальной горизонтальной позиции стрелки

Рендерит контейнеры с нужными стилями относительно циферблата (большой круг)

Если у вас есть доступ к инструментам React Developer Tools для Chrome или Firefox (http://mng.bz/mt5P и http://mng.bz/DANq), откройте панель React в DevTools (или ее аналог в Firefox). У меня выводится информация о том, что элемент `<Clock>` имеет два дочерних элемента (рис. 4.8). Обратите внимание: React DevTools сообщает имена компонентов вместе с состоянием `currentTime`. До чего полезный инструмент для отладки!

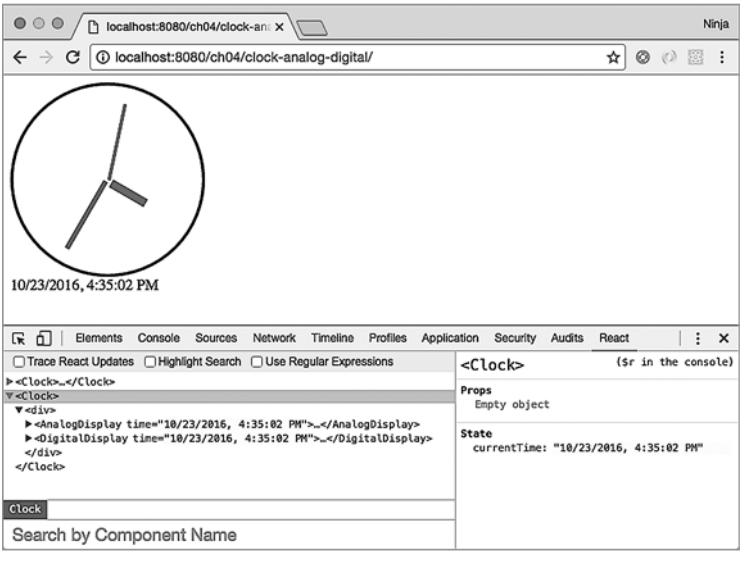


Рис. 4.8. В React DevTools s v0.15.4 выводится информация о двух компонентах

В этом примере я использовал анонимные выражения, хранящиеся в виде `const`-переменных. В другом решении используется синтаксис с объявлениями именованных функций:

```
function AnalogDisplay(props) {...}
```

Также можно воспользоваться объявлением именованной функции по ссылке из переменной:

```
const AnalogDisplay = function AnalogDisplay(props) {...}
```

ОБЪЯВЛЕНИЯ ФУНКЦИЙ В JAVASCRIPT

В JavaScript предусмотрено несколько способов определения функций. Вы можете написать анонимное функциональное выражение, которое тут же используется (как правило, в качестве обратного вызова):

```
function() { return 'howdy'}
```

Также можно создать IIFE (немедленно вызываемую функцию):

```
(function() {
  return('howdy')
})()
```

Ссылка на анонимное функциональное выражение может храниться в переменной:

```
let sayHelloInMandarin = function() { return 'ni hao'}
```

Так выглядит именованное функциональное выражение:

```
function sayHelloInTatar() { return 'sälam'}
```

А это именованное функциональное выражение, ссылка на которое хранится в переменной:

```
let sayHelloInSpanish = function digaHolaEnEspanol() { return 'hola'}
```

Наконец, можно использовать немедленно вызываемое именованное функциональное выражение:

```
(function sayHelloInTexan() {
  return('howdy')
})()
```

Не существует стрелочного синтаксиса для именованных функций.

Как видите, компоненты `AnalogDisplay` и `DigitalDisplay` не имеют состояния. Они также не содержат методов, если не считать тело функции, которое не похоже на `render()` в нормальном определении класса React. Вся логика и состояния приложения находятся в `Clock`.

С другой стороны, в компонентах без состояния размещается только логика анимации, но этот факт тесно связан с аналоговым отображением времени. Разумеется, размещение аналоговой анимации в `Clock` было бы признаком плохой архитектуры. Теперь у вас два компонента, и любой из них (или оба сразу) можно рендерить из `Clock`. Правильное использование компонентов без состояния с несколькими компонентами, обладающими состоянием, позволяет создать более гибкую, простую и качественную архитектуру.

Как правило, когда разработчик React говорит «без состояния», он имеет в виду компонент, созданный с использованием функции или «стрелочного» синтаксиса. Компонент без состояния может быть создан в виде класса, но так поступать не рекомендуется, потому что кто-то другой (или вы через полгода) сможет слишком легко добавить состояние. Нет соблазна — нет возможности усложнить код!

Вероятно, вас интересует, может ли компонент без состояния содержать методы? Разумеется, если вы используете классы — да, они могут содержать методы; но, как я упоминал выше, большинство разработчиков использует функции. И хотя

методы можно присоединять к функциям (в JavaScript они тоже являются объектами), код получится неэлегантным, потому что в функции нельзя использовать `this` (значением будет не компонент, а `window`):

```
// Антипаттерн: не делайте этого.
const DigitalDisplay = function(props) {
  return <div>{DigitalDisplay.locale(props.time)}</div>
}
DigitalDisplay.locale = (time)=>{
  return (new Date(time)).toLocaleString('EU')
}
```

Если вам понадобится выполнить некую логику, связанную с представлением, создайте новую функцию прямо в компоненте без состояния:

```
// Правильный паттерн
const DigitalDisplay = function(props) {
  const locale = time => (new Date(time)).toLocaleString('EU')
  return <div>{locale(props.time)}</div>
}
```

Ваши компоненты без состояния должны быть простыми: ни состояния, ни методов. В частности, в них не должно быть вызовов внешних методов или функций, потому что их результаты могут нарушить предсказуемость (и концепцию чистоты).

4.6. Вопросы

1. Какой синтаксис используется для назначения состояния в методе компонента (не в конструкторе): `this.setState(a)`, `this.state = a` или `this.a = a`?
2. Если вы хотите обновить процесс рендера, обычно для этого выполняется изменение свойств в компонентах: `this.props.a=100`. Да или нет?
3. Состояния являются изменяемыми, а свойства неизменяемы. Да или нет?
4. Компоненты без состояния могут быть реализованы в виде функций. Да или нет?
5. Как определить первые переменные состояния при создании элемента? Что вы используете: `setState()`, `initialState()`, `this.state =...` в конструкторе или `setInitialState()`?

4.7. Итоги

- Состояния изменяемы, а свойства неизменяемы.
- Метод `getInitialState` позволяет компонентам иметь исходный объект состояния.

- Метод `this.setState` обновляет только те свойства, которые были ему переданы, а не все свойства объекта состояния.
- Синтаксис `{}` используется для рендера переменных и выполнения JavaScript в коде JSX.
- Конструкция `this.state.NAME` используется для обращения к переменным состояния.
- При работе с React рекомендуется по возможности использовать компоненты без состояния.

4.8. Ответы

1. `this.setState(a)`, потому что значение `this.state` никогда, *никогда* не должно присваиваться напрямую. Запись `this.a` ничего не делает с состоянием — она только создает поле/атрибут/свойство экземпляра.
2. Нет. Изменение свойства в компоненте не инициирует рендеринг.
3. Да. Невозможно изменить свойство из компонента — только из его родителя. И наоборот, состояние может изменяться только компонентом.
4. Да. Вы можете использовать «стрелочную» функцию или традиционное определение `function() {}`, но оба варианта должны возвращать элемент (один элемент).
5. `this.state = ...` в конструкторе или `getInitialState()` при использовании `createClass()`.

5 События жизненного цикла компонентов React

Посмотрите вступительный видеоролик к этой главе, отсканировав QR-код или перейдя по ссылке http://reactquickly.co/videos/ch05.

ЭТА ГЛАВА ОХВАТЫВАЕТ СЛЕДУЮЩИЕ ТЕМЫ:

- Общие сведения о событиях жизненного цикла компонентов React.
- Категории событий.
- Реализация события.
- События обновления.

В главе 2 было рассказано о том, как создавать компоненты, но в некоторых ситуациях нужны более точные средства контроля над компонентами. Допустим, вы строите нестандартный компонент кнопки-переключателя, размеры которого могут изменяться в зависимости от ширины экрана. А может быть, вы строите меню, которое должно получать информацию от сервера, отправляя запрос XHR.

Один из способов заключается в реализации необходимой логики перед созданием экземпляра компонента и его повторным созданием с другими свойствами. К сожалению, в этом случае не удастся создать автономный компонент, а значит, вы лишитесь преимуществ компонентной архитектуры.

В таких случаях лучше всего воспользоваться событиями жизненного цикла компонентов. Подключая события, вы можете внедрить необходимую логику в компоненты. Более того, вы можете использовать другие события для расширения возможностей компонентов и включить специальную логику относительно того, нужно ли повторно рендерить их представления (переопределяя алгоритм React по умолчанию).

Возвращаясь к примерам с нестандартным переключателем и меню, кнопка может присоединять слушатели событий к окну (`onResize`) при создании компонента кнопки и отсоединять их при удалении компонента. А меню может загружать данные с сервера при подключении (вставке) элемента React в реальную модель DOM.

ПРИМЕЧАНИЕ Исходный код примеров этой главы доступен по адресам www.manning.com/books/react-quickly и https://github.com/azat-co/react-quickly/tree/master/ch05 (в папке ch05 репозитория GitHub https://github.com/azat-co/react-quickly). Также некоторые демонстрационные примеры доступны по адресу http://reactquickly.co/demos.

5.1. Общие сведения о событиях жизненного цикла компонентов React

React предоставляет возможность управлять поведением компонента и настраивать его на основании событий его жизненного цикла. Эти события делятся на следующие категории:

- *События подключения* — происходят при присоединении элемента React (экземпляра компонента класса) к узлу DOM.
- *События обновления* — происходят при обновлении элемента React в результате изменения значений его свойств или состояния.
- *События отключения* — происходят при отсоединении элемента React от модели DOM.

У каждого без исключения компонента React есть события жизненного цикла, которые срабатывают в определенные моменты в зависимости от того, что сделал или собирается сделать компонент. Некоторые из них выполняются всего один раз, другие продолжают выполняться снова и снова.

События жизненного цикла позволяют реализовать нестандартную логику, которая расширяет возможности компонентов. С их помощью также можно изменять поведение компонентов: например, принимая решение о том, когда выполнить повторный рендеринг. Это повышает быстродействие из-за исключения лишних операций. Другие варианты использования — получение данных от серверной части или интеграция с событиями DOM или другими библиотеками клиентской части. А теперь повнимательнее посмотрим, как работают категории событий, какие события в них входят и в какой последовательности эти события выполняются.

5.2. Категории событий

React определяет несколько событий компонентов в трех категориях (рис. 5.1 и табл. 5.1 далее в этой главе). Каждая категория может инициировать события переменное количество раз:

- *Подключение* — React инициирует события только один раз.
- *Обновление* — React может инициировать события многократно.
- *Отключение* — React инициирует события только один раз.

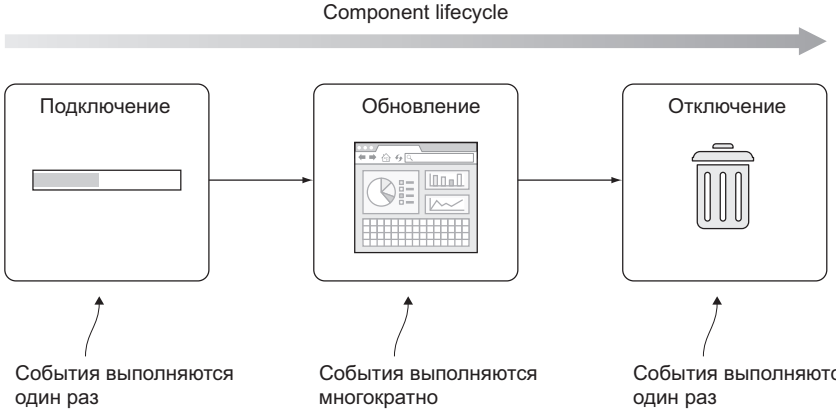


Рис. 5.1. Категории событий жизненного цикла в ходе существования компонента и количество возможных срабатываний событий в каждой категории

Кроме событий жизненного цикла я включу `constructor()` для демонстрации порядка выполнения от начала до конца за время жизненного цикла компонента (обновление может происходить многократно):

- `constructor()` — выполняется при создании элемента; позволяет задать свойства по умолчанию (глава 2) и исходное состояние (глава 4).
- *Подключение*:
  - `componentWillMount()` — происходит перед подключением к DOM.
  - `componentDidMount()` — происходит после подключения и рендера.
- *Обновление*:
  - `componentWillReceiveProps(nextProps)` — происходит перед получением свойств компонентом.
  - `shouldComponentUpdate(nextProps, nextState)-> bool` — позволяет оптимизировать повторный рендеринг компонента, помогая вам определить, когда нужно или не нужно проводить обновление.
  - `componentWillUpdate(nextProps, nextState)` — происходит непосредственно перед обновлением компонента.
  - `componentDidUpdate(prevProps, prevState)` — происходит сразу же после обновления компонента.

- *Отключение*:
  - `componentWillUnmount function()` — позволяет отключить любые слушатели событий или провести любые завершающие действия перед отключением компонента.

Обычно разработчик может четко определить, когда инициируется событие, по его имени. Например, событие `componentDidUpdate()` инициируется при обновлении компонента. В других случаях существуют тонкие различия. В табл. 5.1 приведена последовательность событий жизненного цикла (сверху вниз) и зависимости некоторых из них от изменений свойств или состояния (столбцы «Свойства компонента» и «Состояние компонента»).

Таблица 5.1. События жизненного цикла (и их зависимость от состояний и свойств)

Подключение	Обновление свойств компонента	Обновление состояния компонента	Обновление с использованием forceUpdate()	Отключение
`constructor()`				
`componentWill-Mount()`				
	`componentWill-ReceiveProps()`			
	`shouldComponent-Update()`	`shouldComponent-Update()`		
	`componentWill-Update()`	`componentWill-Update()`	`componentWill-Update()`	
	`render()`	`render()`	`render()`	
	`componentDid-Update()`	`componentDid_ Update()`	`componentDid-Update()`	
`componentDid-Mount()`				
				`componentWill-Unmount()`

Существует еще одна ситуация, в которой компонент может быть отрендерен: при вызове `this.forceUpdate()`. Как можно догадаться по имени, этот метод инициирует принудительное обновление. Возможно, вам придется воспользоваться им, когда обновление состояния или свойств почему-то не приводит к рендерингу. Например, это может произойти в том случае, если в `render()` используются данные, которые не относятся к состоянию или свойствам, и эти данные изменяются, а значит, обновление придется инициировать вручную. В общем случае (и по мнению команды разработчиков React) лучше обойтись без использования метода `this.`

`forceUpdate()` (http://mng.bz/v5sU), так как он нарушает чистоту компонентов (см. следующую врезку). А теперь определим событие, чтобы увидеть его в действии.

ЧИСТЫЕ ФУНКЦИИ

В компьютерной теории вообще — не только в React — «чистой» называется функция, которая:

- Для одинаковых входных данных всегда возвращает один результат.
- Не имеет побочных эффектов (то есть не изменяет внешние состояния).
- Не зависит от внешних состояний.

Например, следующая чистая функция удваивает значение входного значения: `f(x) = 2x` или в JavaScript/Node `let f= (n)=>2*n`. Вот как она выглядит в действии:

```
let f = (n)=>2*n
consoleg.log(f(7))
```

Нечистая функция для удвоения чисел выглядит так (добавление фигурных скобок избавляет от неявного возвращения однострочной «стрелочной» функции):

```
let sharedStateNumber = 7
let double
let f = ()=> {double =2*sharedStateNumber}
f()
console.log(double)
```

Чистые функции — краеугольный камень функционального программирования (ФП), которое сводит к минимуму состояние настолько, насколько это возможно. Разработчики (и особенно функциональные программисты) предпочитают чистые функции прежде всего из-за того, что их использование избавляет от проблем использования общего состояния, а это упрощает разработку и способствует разделению разных фрагментов логики. Кроме того, чистые функции упрощают тестирование. Что касается React, вы уже знаете, что чем больше компонентов без состояния и чем меньше зависимостей, тем лучше, — вот почему разработчикам следует по возможности создавать чистые функции.

В некоторых отношениях ФП противоречит канонам ООП (или, может, ООП противоречит канонам ФП?); поклонники ФП говорят, что Fortran и Java были тупиковыми ветвями программирования, а программировать нужно на Lisp (на Clojure и Elm в наши дни). Следить за этими спорами — одно удовольствие. Лично я слегка склоняюсь к функциональному подходу.

На тему ФП написано много хороших книг, потому что концепция существует уже несколько десятилетий. Я не стану углубляться в подробности, но настоятельно порекомендую побольше узнать о ФП, потому что это повысит вашу квалификацию как программиста, даже если вы никогда не будете применять ФП в своей работе.

5.3. Реализация события

Чтобы реализовать события жизненного цикла, определите их в классе как методы (см. раздел 3.2.5) — React ожидает, что вы будете соблюдать это соглашение. React проверяет, существует ли метод с именем события; обнаружив такой метод, React вызывает его. В противном случае React продолжает нормальное выполнение. Разумеется, в именах событий учитывается регистр символов, как и в любом имени JavaScript.

На происходящее можно взглянуть и под другим углом. React вызывает некоторые методы во время жизненного цикла компонента, если эти методы определены. Например, если вы определили метод `componentDidMount()`, то React будет вызывать этот метод при подключении элемента этого класса компонента. Метод `componentDidMount()` относится к категории событий подключения из табл. 5.1 и будет вызываться по одному разу для каждого экземпляра класса компонента:

```
class Clock extends React.Component {
componentDidMount() {
}
  ...
}
```

Если метод `componentDidMount()` не определен, React не будет исполнять никакой код для этого события. Итак, имя метода должно совпадать с именем события. С этого момента я будут использовать термины «событие», «обработчик события» и «метод» как синонимы.

Как вы, вероятно, догадались по названию, метод `componentDidMount()` вызывается при вставке компонента в DOM. В этом методе рекомендуется размещать код для интеграции с другими фреймворками клиентской части и библиотеками, а также код отправки запросов XHR-серверу, потому что в этой точке жизненного цикла элемент компонента находится в реальной модели DOM, а вы получаете доступ ко всем его элементам, включая дочерние.

А теперь вернемся к проблемам, о которых я упоминал в начале главы: изменение размеров и загрузка данных с сервера. Для первого можно создать слушателя события в `componentDidMount()`, который будет прослушивать события `window.resize`, для второго — выдать вызов XHR в `componentDidMount()` и обновить состояние при получении ответа от сервера.

Что не менее важно, метод `componentDidMount()` удобен в изоморфном/универсальном коде (когда одни и те же компоненты используются на сервере и в браузере). Вы можете разместить в этом методе логику, относящуюся исключительно к браузеру, и быть уверенными в том, что она будет вызываться только для рендера в браузере, а не на стороне сервера. Изоморфный код JavaScript с React рассматривается в главе 16.

Многие разработчики лучше всего учатся на примерах. Рассмотрим тривиальный случай с использованием `componentDidMount()` для вывода информации DOM в консоль. Это возможно, потому что это событие срабатывает после завершения всего рендеринга, а следовательно, будут доступны все элементы DOM.

Создание слушателей событий для событий жизненного цикла компонента проходит тривиально: вы определяете метод для компонента/класса. Просто для интереса добавим метод `componentWillMount()`, чтобы продемонстрировать отсутствие информации реальной модели DOM для этого элемента на этой стадии.

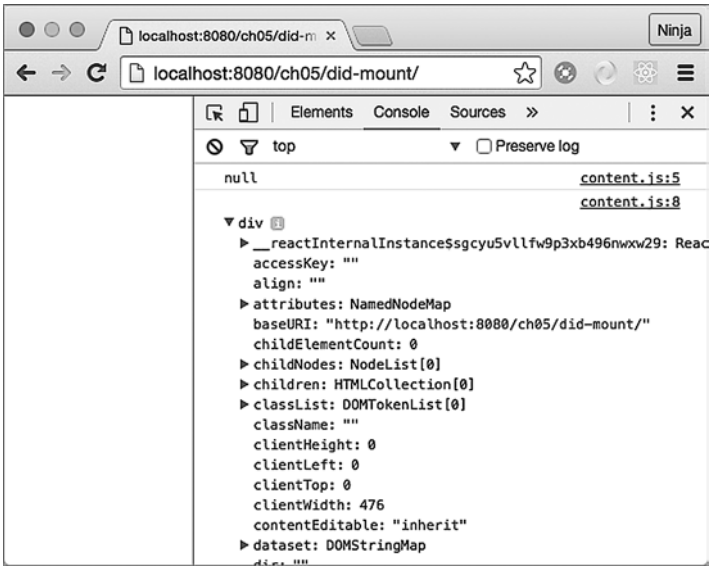


Рис. 5.2. Во второй записи выводится узел DOM, потому что метод componentDidMount() сработал, когда элемент уже был сгенерирован и подключен к реальной модели DOM

Для получения информации узлов DOM используется вспомогательная функция React DOM `ReactDOM.findDOMNode()`, которой передается класс. Обратите внимание: `DOM` записывается не в «верблюжьем регистре», а только символами верхнего регистра:

```
class Content extends React.Component {
  componentWillMount() {
    console.log(ReactDOM.findDOMNode(this))
  }
  componentDidMount() {
    console.dir(ReactDOM.findDOMNode(this))
  }
  render() {
```

Узел DOM должен быть неопределенным (null)

**Узел DOM должен содержать **

```
    return (

    )
  }
}
```

В консоль разработчика выводится следующий результат, по которому можно убедиться в том, что при выполнении метода `componentDidMount()` уже существуют реальные элементы DOM (рис. 5.2):

```
html
null
div
```

5.4. Сразу все действия

В листинге 5.1 (ch05/logger/jsx/content.jsx) и листинге 5.2 (ch05/logger/jsx/logger.jsx) продемонстрированы сразу все события. А пока достаточно знать, что они похожи на классы в том смысле, что они позволяют повторно использовать код. Эта примесь (mixin) может пригодиться для отладки; она выводит все события, свойства и состояние, когда компонент готовится к повторной перерисовке, и после того, как он будет повторно перерисован.

Листинг 5.1. Троекратный рендер и обновление компонента Logger

```
class Content extends React.Component {
  constructor(props) {
    super(props)
    this.launchClock()
    this.state = {
      counter: 0,
      currentTime: (new Date()).toLocaleString()
    }
  }
  launchClock() {
    setInterval(()=>{
      this.setState({
        counter: ++this.state.counter,
        currentTime: (new Date()).toLocaleString()
      })
    }, 1000)
  }
  render() {
    if (this.state.counter > 2) return
    return <Logger time="{this.state.currentTime}"></Logger>
  }
}
```

Листинг 5.2. Отслеживание событий жизненного цикла компонента

```
class Logger extends React.Component {
  constructor(props) {
    super(props)
    console.log('constructor')
  }
  componentWillMount() {
    console.log('componentWillMount is triggered')
  }
  componentDidMount(e) {
    console.log('componentDidMount is triggered')
    console.log('DOM node: ', ReactDOM.findDOMNode(this))
  }
  componentWillReceiveProps(newProps) {
    console.log('componentWillReceiveProps is triggered')
    console.log('new props: ', newProps)
  }
  shouldComponentUpdate(newProps, newState) {
    console.log('shouldComponentUpdate is triggered')
    console.log('new props: ', newProps)
    console.log('new state: ', newState)
    return true
  }
  componentWillUpdate(newProps, newState) {
    console.log('componentWillUpdate is triggered')
    console.log('new props: ', newProps)
    console.log('new state: ', newState)
  }
  componentDidUpdate(oldProps, oldState) {
    console.log('componentDidUpdate is triggered')
    console.log('new props: ', oldProps)
    console.log('old props: ', oldState)
  }
  componentWillUnmount() {
    console.log('componentWillUnmount')
  }
  render() {
    // console.log('rendering... Display')
    return (
      {this.props.time}
    )
  }
}
```

Функции и события жизненного цикла из компонента `Display` выводят данные на консоль при выполнении этой веб-страницы. Не забудьте открыть консоль в браузере, потому что весь вывод происходит именно здесь, как показано на рис. 5.3!

Как упоминается в тексте и показано на рисунке, событие монтирования срабатывает только один раз. Это хорошо видно по результатам в журнале. После того как счетчик в `Context` достигнет 3, функция `render` не будет использовать `Display` и компонент отключается (рис. 5.4).

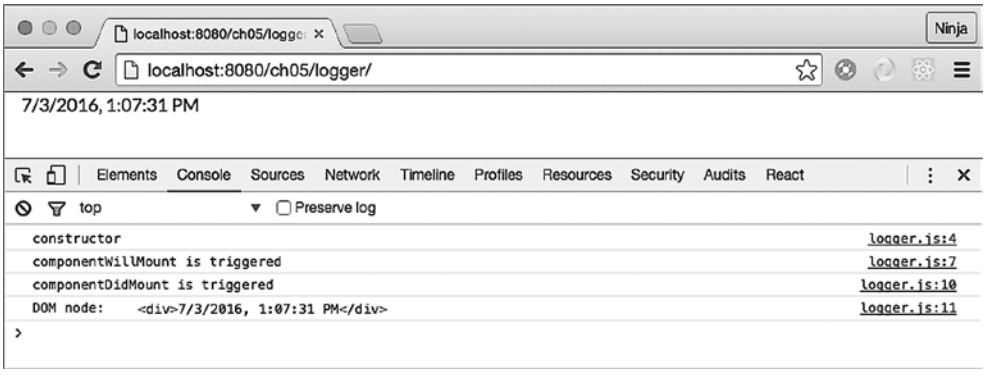


Рис. 5.3. Компонент подключен

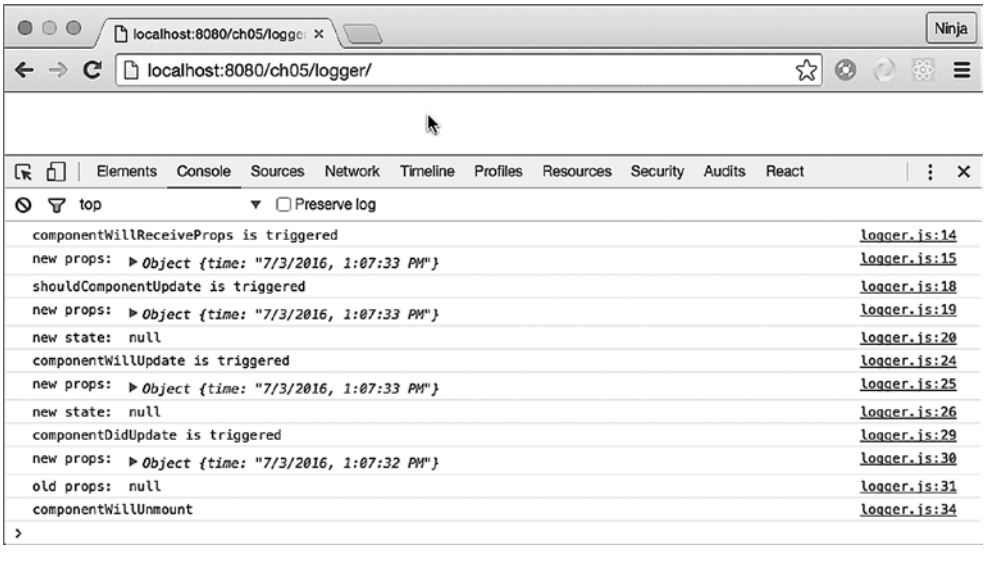


Рис. 5.4. Контент удаляется через 2 секунды; следовательно, componentWillUnmount() выводится непосредственно перед удалением

Зная о событиях жизненного цикла компонента, вы сможете использовать их для реализации логики компонентов, например загрузки данных.

5.5. События подключения

Категория событий подключения относится к присоединению компонентов к реальной модели DOM. Монтирование может рассматриваться как процесс появления элементов React в модели DOM. Обычно это происходит при использовании

компонента в `ReactDOM.render()` или в методе `render()` другого компонента более высокого порядка, который был отрендерен в DOM.

К категории подключения относятся следующие события:

- `componentWillMount()` — React знает, что этот элемент появится в реальной модели DOM.
- `componentDidMount()` — элемент React был «вставлен» в реальную модель DOM и стал узлом DOM.

Выполнение `constructor()` происходит до `componentWillMount()`. Кроме того, React сначала рендерит, а затем подключает элементы. (Рендер в этом контексте означает вызов метода `render()` класса, а не отображение DOM.) События в интервале от `componentWillMount()` до `componentDidMount()` перечислены в табл. 5.1.

5.5.1. componentWillMount()

Стоит упомянуть, что метод `componentWillMount()` вызывается только один раз за жизненный цикл компонента. Время выполнения — непосредственно перед исходным рендерингом.

Событие жизненного цикла `componentWillMount()` выполняется при рендере элемента React в браузере вызовом `ReactDOM.render()`. Считайте это присоединением (или подключением) элемента React к реальному узлу DOM. Это происходит в браузере, то есть в клиентской части.

Если вы рендерите компонент React на сервере (в серверной части с использованием изоморфного/универсального кода JavaScript; см. главу 16), который фактически получает строку HTML, то это событие также будет инициировано, хотя на сервере нет ни DOM, ни подключения в данном случае!

В главе 4 было показано, как обновить состояние `currentTime` с использованием `Date` и `setInterval()`. Серия обновлений была запущена в `constructor()` вызовом `launchClock()`. Это также можно сделать в коде `componentWillMount()`.

Изменение состояния обычно инициирует повторный рендеринг, верно? В то же время если вы обновили состояние вызовом `setState()` в методе `componentWillMount()` или инициировали обновления, как это было сделано с `Clock`, то `render()` получит обновленное состояние. Самое замечательное здесь то, что даже если новое состояние отличается от прежнего, повторный рендеринг не потребуется, потому что `render()` получит новое состояние. Другими словами, вы можете вызвать `setState()` в `componentWillMount()`. `render()` получит новые значения, если они есть, и дополнительный повторный рендеринг не потребуется.

5.5.2. componentDidMount()

С другой стороны, `componentDidMount()` вызывается после исходного рендеринга. Код выполняется только один раз и только в браузере, не на сервере. Это может

быть удобно, когда вам нужно реализовать код, выполняемый только в браузерах (например, запрос XHR).

В этом событии жизненного цикла доступны любые ссылки на дочерние элементы (например, для обращения к соответствующему представлению DOM). Обратите внимание на то, что метод `componentDidMount()` дочерних компонентов вызывается раньше одноименного метода родительских компонентов.

Как упоминалось ранее, событие `componentDidMount()` лучше всего подходит для интеграции с другими библиотеками JavaScript. Например, вы можете получить данные в формате JSON, содержащие список пользователей с их данными. Затем эту информацию можно вывести с использованием таблицы Twitter Bootstrap для получения страницы, показанной на рис. 5.5.

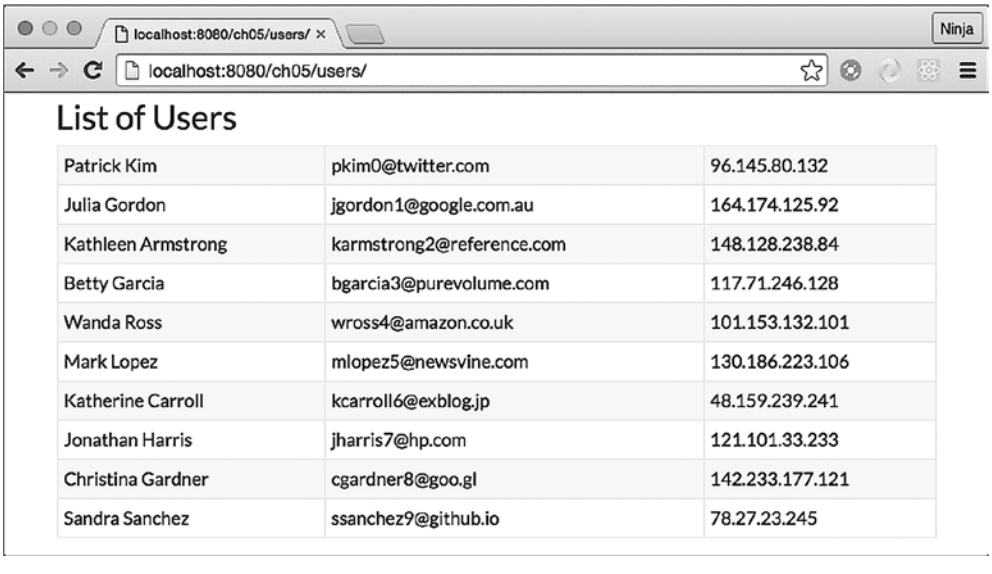


Рис. 5.5. Вывод списка пользователей (полученного из хранилища данных) со стилевым оформлением Twitter Bootstrap

Проект имеет следующую структуру:

```
/users
  /css
    bootstrap.css
  /js
    react.js
    react-dom.js
    script.js
    - users.js
  /jsx
```

```
    script.jsx
    users.jsx
  index.html
  real-user-data.json
```

В событии элемент DOM уже существует, и вы можете отправлять запросы XHR/AJAX для загрузки данных с использованием новой API-функции fetch():

```
fetch(this.props['data-url'])
  .then((response)=>response.json())
  .then((users)=>this.setState({users: users}))
```

FETCH API

Fetch API (http://mng.bz/mbMe) позволяет выдавать запросы XHR с использованием обещаний (promises). Этот интерфейс доступен в большинстве современных браузеров, но вам стоит проверить спецификации (https://fetch.spec.whatwg.org) и стандарт (https://github.com/whatwg/fetch), чтобы узнать, реализован ли он в браузерах, которые должны поддерживаться вашими приложениями. Использование `fetch()` достаточно тривиально — вы передаете URL и определяете столько обещаний `.then`, сколько требуется:

```
fetch('http://node.university/api/credit_cards/')
  .then(function(response) {
    return response.blob()
  })
  .then(function(blob) {
    // Обработка блока двоичных данных
  })
  .catch(function(error) {
    console.log('A problem with your fetch operation: ' +
      error.message)
  })
```

Если браузер, который вы разрабатываете, не поддерживает `fetch()`, для него можно использовать оболочку совместимости (shim) или воспользоваться любой другой агентской библиотекой HTTP — например, superagent (https://github.com/visionmedia/super-agent), request (https://github.com/request/request), axios (https://github.com/mzabriskie/axios), или даже `$.get()` или `$.ajax()` (http://api.jquery.com/jquery.ajax) из jQuery.

Вы можете поместить запрос на загрузку XHR в `componentDidMount()`. Можно подумать, что размещение кода в `componentWillMount()` оптимизирует загрузку, но здесь возникают две проблемы: если вы получите данные с сервера быстрее, чем завершится рендер, вы можете инициировать рендер для неподключенного элемента с непредвиденными последствиями. Кроме того, если вы собираетесь планировать использовать компонент на сервере, то сработает также `componentWillMount()`.

А теперь рассмотрим весь компонент с загрузкой данных в `componentDidMount()` (ch05/users/jsx/users.jsx).

Листинг 5.3. Загрузка данных для отображения в таблице

```
class Users extends React.Component {
  constructor(props) {
    super(props)
    this.state = {
      users: []
    }
  }
  componentDidMount() {
    fetch(this.props['data-url'])
      .then((response)=>response.json())
      .then((users)=>this.setState({users: users}))
  }
  render() {
    return <div className="container">
    <h1>List of Users</h1>
    <table className="table-striped table-condensed table table-bordered
     ➥ table-hover">
      <tbody>{this.state.users.map((user)=>
        <tr key={user.id}>
          <td>{user.first_name} {user.last_name}</td>
          <td> {user.email}</td>
          <td> {user.ip_address}</td>
        </tr>)}
      </tbody>
    </table>
  </div>
  }
}
```

Инициализирует состояние пустым массивом

Выполняет GET-запрос XHR с использованием URL из свойства для загрузки данных

Загружает информацию из ответа и присваивает ее состоянию

Перебирает данные состояния и строит строки таблицы

Обратите внимание: в конструкторе `users` присваивается пустой массив (`[]`). Это избавляет от необходимости проверки существования позднее в `render()`. Повторные проверки и ошибки из-за неопределенных значений — самый верный способ попусту потратить время и ввести кучу лишних символов с клавиатуры. Присваивание исходных значений позволит избежать значительных неприятностей в будущем! Иначе говоря, следующий пример является антипаттерном:

```
// Антипаттерн: не пытайтесь повторить!
class Users extends React.Component {
  constructor(props) {
    super(props)
  }
  ...
  render() {
    return <div className="container">
```

Пустой массив не присваивается изначально

```
      <h1>List of Users</h1>
      <table className="table-striped table-condensed table table-bordered
       ➥ table-hover">
        <tbody>{(this.state.users && this.state.users.length>0) ?
          this.state.users.map((user)=>
          <tr key={user.id}>
            <td>{user.first_name} {user.last_name}</td>
            <td> {user.email}</td>
            <td> {user.ip_address}</td>
          </tr>) : ''}
        </tbody>
      </table>
    </div>
  }
}
```

Проверка существования (не нужна с исходным присваиванием)

5.6. События обновления

Как упоминалось выше, события подключения часто используются для интеграции React с внешним миром: другими фреймворками, библиотеками или хранилищами данных. События обновления связываются с обновлениями компонентов. Эти события перечислены ниже, от начала до конца жизненного цикла компонента (события жизненного цикла обновления представлены в табл. 5.2, а все события перечислены в табл. 5.1).

1. `componentWillReceiveProps(newProps)`
2. `shouldComponentUpdate()`
3. `componentWillUpdate()`
4. `componentDidUpdate()`

Таблица 5.2. События жизненного цикла, вызываемые по обновлениям компонентов

Обновление свойств компонентов	Обновление состояния компонентов	Обновление с использованием forceUpdate()
`componentWillReceiveProps()`		
`shouldComponentUpdate()`	`shouldComponentUpdate()`	
`componentWillUpdate()`	`componentWillUpdate()`	`componentWillUpdate()`
`render()`	`render()`	`render()`
`componentDidUpdate()`	`componentDidUpdate()`	`componentDidUpdate()`

5.6.1. componentWillReceiveProps(newProps)

Событие `componentWillReceiveProps(newProps)` инициируется при получении компонентом новых свойств. Эта стадия называется *предстоящим переходом свойства*. Данное событие позволяет вмешаться в работу компонента и включить новую логику между получением новых свойств и перед запуском `render()`.

Метод `componentWillReceiveProps(newProps)` получает новые свойства в аргументе. Он не вызывается при исходном рендеринге компонента. Этот метод пригодится в тех случаях, когда вы хотите сохранить новое свойство и задать состояние перед повторным рендерингом. Старое значение свойства хранится в объекте `this.props`. Например, следующий фрагмент задает состояние прозрачности, которое в CSS равно 0 или 1 в зависимости от значения логического свойства `isVisible` (1 = `true`, 0 = `false`):

```
componentWillReceiveProps(newProps) {
  this.setState({
    opacity: (newProps.isVisible)?1:0
  })
}
```

В общем случае метод `setState()` в `componentWillReceiveProps(newProps)` не инициирует дополнительный повторный рендеринг.

Несмотря на получение новых свойств, эти свойства не обязательно имеют новые значения (то есть значения, отличные от текущих значений свойств), потому что React не может определить, изменились значения свойств или нет. Следовательно, `componentWillReceiveProps(NewProps)` вызывается каждый раз при повторном рендеринге (родительской структуры или по вызову), независимо от изменений свойств/значений. А следовательно, нельзя считать, что `newProps` всегда имеет значения, отличные от текущих значений свойств.

В то же время повторный рендеринг (вызовом `render()`) не обязательно означает изменения в реальной модели DOM. Решение о том, нужно ли обновлять реальную модель DOM и что именно должно обновляться, делегируется `shouldComponentUpdate()` и процессу согласования[1].

5.6.2. shouldComponentUpdate()

Затем идет событие `shouldComponentUpdate()`, которое инициируется непосредственно перед рендером. Рендерингу предшествует получение новых свойств или состояния. Событие `shouldComponentUpdate()` не инициируется для исходного рендеринга или для `forceUpdate()` (см. табл. 5.1).

---

[1] Другие причины, по которым React не может выполнить более умные проверки перед вызовом componentWillReceiveProps(newProps), представлены в обширной статье «(A => B) !=> (B => A)» Джима Спрока (Jim Sproch), React, 8 января 2016 г., http://mng.bz/3WpG.

Вы можете реализовать событие `shouldComponentUpdate()` с `return false`, чтобы запретить React повторный рендеринг. Например, это может быть полезно, когда проверка показывает, что изменений нет, и вы хотите избежать лишних потерь быстродействия (при сотнях компонентов). Например, следующий фрагмент использует бинарный оператор `+` для преобразования логического значения `isVisible` в число и сравнения его со значением `opacity`:

```
shouldComponentUpdate(newProps, newState) {
  return this.state.opacity !== + newProps.isVisible
}
```

Если значение `isVisible` ложно, а состояние `this.state.opacity` равно 0, то весь вызов `render()` пропускается; кроме того, события `componentWillUpdate()` и `componentDidUpdate()` не вызываются. Фактически вы можете управлять тем, будет компонент повторно отрендерен или нет.

5.6.3. componentWillUpdate()

Что касается `componentWillUpdate()`, этот метод вызывается непосредственно перед рендером, а ему предшествует получение новых свойств или состояния. Он не вызывается для исходного рендеринга. Используйте метод `componentWillUpdate()` для выполнения подготовительных действий перед обновлением и старайтесь не использовать `this.setState()` в этом методе! Почему? Представьте, что вы пытаетесь инициировать обновление в то время, когда компонент находится в процессе обновления. Добром это не кончится!

Если `shouldComponentUpdate()` вернет `false`, то `componentWillUpdate()` не инициируется.

5.6.4. componentDidUpdate()

Событие `componentDidUpdate()` инициируется сразу же после того, как обновление компонентов будет отражено в DOM. И снова этот метод не вызывается для исходного рендеринга. `componentDidUpdate()` используется для написания кода, который работает с DOM и другими элементами после того, как компонент будет обновлен, потому что на этой стадии вы получите все обновления, отрендеренные в DOM.

Каждый раз, когда к модели DOM что-то подключается или обновляется, должен существовать способ это «что-то» отключить. Следующее событие предоставляет место для размещения логики отключения.

5.7. События отключения

В React под «отключением» понимается отсоединение или удаление элемента из DOM. К этой категории относится только одно событие, и она является последней категорией в жизненном цикле компонента.

5.7.1. componentWillUnmount()

Событие `componentWillUnmount()` вызывается непосредственно перед отключением компонента от DOM. Вы также можете включить в этот метод все необходимые завершающие действия, например: остановку таймеров, очистку любых элементов DOM или отсоединение событий, созданных в `componentDidMount`.

5.8. Простой пример

Предположим, вам поручено создать веб-приложение Note (для сохранения текста в интернете). Вы реализовали компонент, но первые пользователи сообщают, что при случайном закрытии окна (или вкладки) вся текущая работа теряется. Давайте реализуем диалоговое окно подтверждения на рис. 5.6.

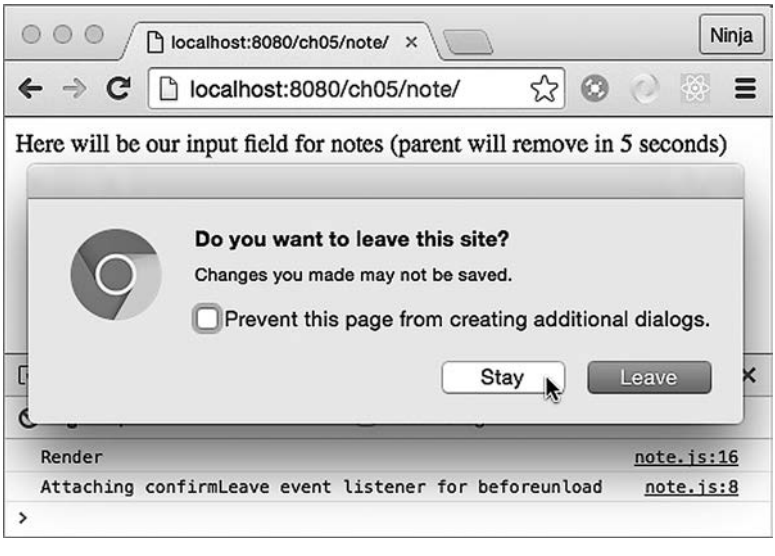


Рис. 5.6. Диалоговое окно для подтверждения ухода пользователя со страницы

Для реализации такого диалогового окна необходимо прослушивать специальное событие окна. Вся хитрость здесь связана с очисткой после того, как элемент станет ненужным, потому что если элемент будет удален, а событие останется, это может привести к утечке памяти! Лучшее решение проблемы — присоединение события при подключении компонента и его отсоединение при отключении.

Структура проекта выглядит так:

```
/note
  /jsx
    note.jsx
```

```
    script.jsx
  /js
    note.jsx
    react.js
    react-dom.js
    script.js
  index.html
```

«Родное» событие браузера `window.onbeforeunload` (с дополнительным кодом межбраузерной поддержки) выглядит прямолинейно:

```
window.addEventListener('beforeunload',function () {
  let confirmationMessage = 'Do you really want to close?'
  e.returnValue = confirmationMessage // Gecko, Trident, Chrome 34+
  return confirmationMessage          // Gecko, WebKit, Chrome < 34
})
```

Следующий подход тоже работает:

```
window.onbeforeunload = function () {
  ...
  return confirmationMessage
}
```

Поместим этот код в слушателя события в `componentDidMount()`; слушатель события будет удаляться в `componentWillUnmount()` (ch05/note/jsx/note.jsx).

Листинг 5.4. Добавление и удаление слушателя события

```
class Note extends React.Component {
  confirmLeave(e) {
    let confirmationMessage = 'Do you really want to close?'
    e.returnValue = confirmationMessage // Gecko, Trident, Chrome 34+
    return confirmationMessage // Gecko, WebKit, Chrome <34
  }
  componentDidMount() {
    console.log('Attaching confirmLeave event listener for beforeunload')
    window.addEventListener('beforeunload', this.confirmLeave)
  }
  componentWillUnmount() {
    console.log('Removing confirmLeave event listener for beforeunload')
    window.removeEventListener('beforeunload', this.confirmLeave)
  }
  render() {
    console.log('Render')
    return Here will be our input field for notes (parent will remove in
    ➥ {this.props.secondsLeft} seconds)
  }
}
```

Нужно проверить, как работает ваш код при удалении элемента Note, верно? Для этого нужно удалить элемент Note, чтобы он был отключен от DOM. А следователь-

но, следующим шагом должна стать реализация родителя, в котором вы не только создаете элемент Note, но и удаляете его. Для этого мы воспользуемся таймером (`setInterval()` вам в помощь!), как показано в листинге 5.5 (ch05/note/jsx/script.jsx) и на рис. 5.7.

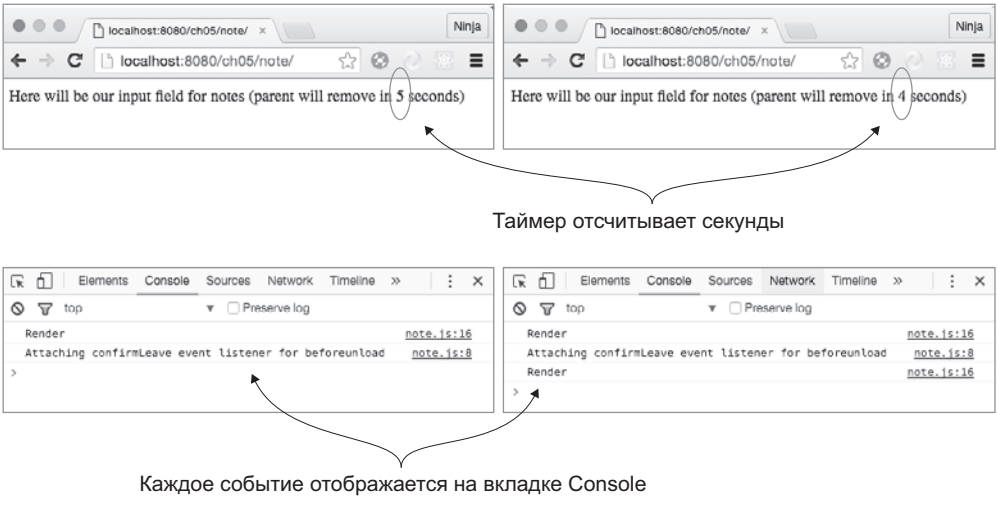


Рис. 5.7. Элемент Note будет заменен другим элементом через 5, 4, … секунд

Листинг 5.5. Рендеринг Note перед удалением

```
let secondsLeft = 5

let interval = setInterval(()=>{
  if (secondsLeft == 0) {
    ReactDOM.render(
      <div>
        Note was removed after {secondsLeft} seconds.
      </div>,
      document.getElementById('content')
    )
    clearInterval(interval)
  } else {
    ReactDOM.render(
      <div>
        <Note secondsLeft={secondsLeft}/>
      </div>,
      document.getElementById('content')
    )
  }
  secondsLeft--
}, 1000)
```

На рис. 5.8 показан результат (с консольным выводом): `render`, присоединение слушателя события, `render` еще 4 раза и удаление слушателя события.

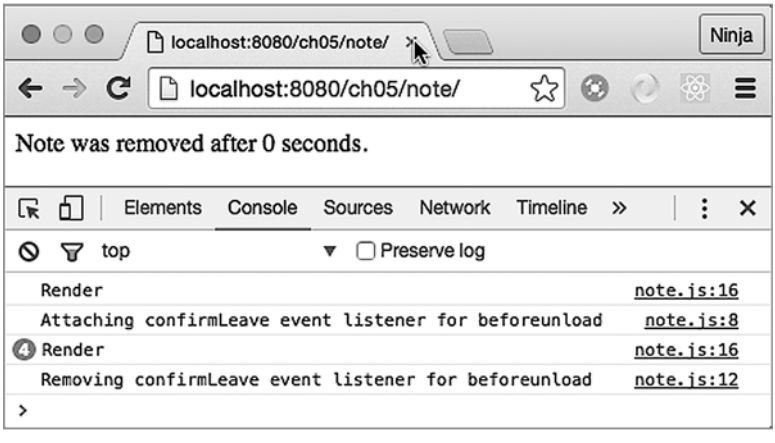


Рис. 5.8. Элемент Note заменяется элементом div, и при попытке покинуть страницу диалоговое окно не отображается

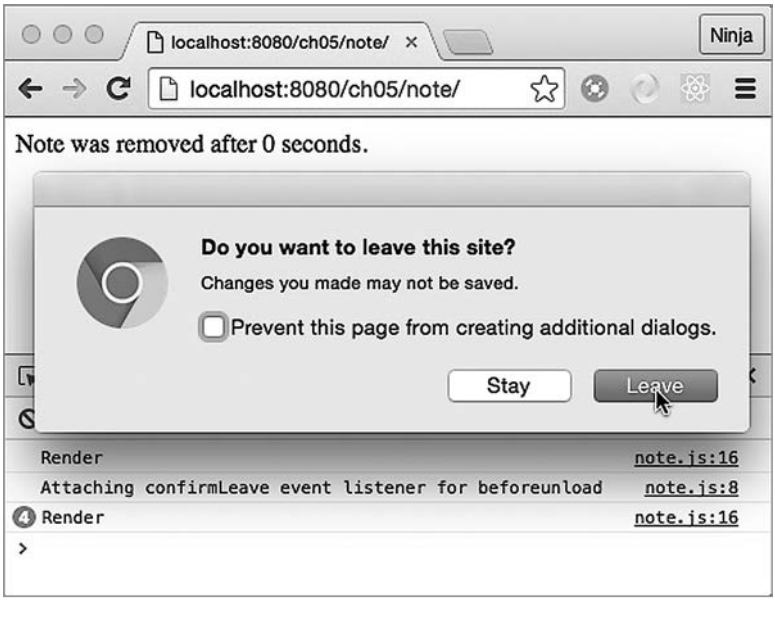


Рис. 5.9. Диалоговое окно подтверждения, которое выводится при попытке пользователя покинуть страницу

Если не удалить слушателя события в `componentWillUnmount()` (попробуйте закомментировать этот метод и посмотрите сами), диалоговое окно останется на странице даже после того, как элемент Note давно исчезнет (рис. 5.9). Такое поведение нежелательно с точки зрения взаимодействия с пользователем и может стать причиной ошибок. Вы можете использовать это событие жизненного цикла для выполнения очистки (завершающих действий) после компонентов.

Команда React прислушивается к обратной связи, полученной от разработчиков. Многие события жизненного цикла позволяют разработчикам настроить поведение своих компонентов. Считайте, что события жизненного цикла — инструмент ниндзя-джедаев-черных поясов от программирования. Программировать можно и без них, но сколько же новых возможностей они откроют перед вами! Интересно, что в отношении эффективных методов программирования и их практического применения все еще идут дискуссии. React продолжает развиваться, и в будущем в области событий жизненного цикла возможны изменения или добавления. Если вам понадобится официальная документация, она доступна по адресу https://facebook.github.io/react/docs/react-component.html.

5.9. Вопросы

1. Метод `componentWillMount()` отработает на сервере. Да или нет?
2. Какое событие сработает первым: `componentWillMount()` или `componentDidMount()`?
3. Какой из методов в следующем списке хорошо подходит для размещения вызова AJAX к серверу с целью получения данных для компонента: `componentWillUnmount()`, `componentHasMounted()`, `componentDidMount()`, `componentWillReceiveData()` или `componentWillMount()`?
4. Вызов `componentWillReceiveProps()` означает, что у элемента произошел повторный рендеринг (из родительской структуры), и вы можете быть уверены в том, что свойства получили новые значения. Да или нет?
5. События подключения происходят многократно при каждом повторном рендеринге. Да или нет?

5.10. Итоги

- Метод `componentWillMount()` вызывается как на стороне сервера, так и на стороне клиента, тогда как `componentDidMount()` вызывается только на стороне клиента.
- События подключения обычно используются для интеграции React с другими библиотеками и получения данных из хранилищ или серверов.

- Метод `shouldComponentUpdate()` используется для оптимизации рендеринга.
- Метод `componentWillReceiveProps()` используется для изменения состояния новыми свойствами.
- События отключения обычно используются для выполнения завершающих действий.
- События обновления предоставляют место для размещения логики, зависящей от новых свойств или состояния. Кроме того, они позволяют более точно управлять тем, в какой момент должно обновляться представление.

5.11. Ответы

1. Да. Несмотря на отсутствие модели DOM, это событие будет инициировано при рендеринге на сервере, тогда как событие `componentDidMount()` инициировано не будет.
2. Сначала `componentWillMount()`, затем `componentDidMount()`.
3. `componentDidMount()`, потому что это событие не будет инициировано на сервере.
4. Нет. Новые значения не гарантированы — React не знает, изменились значения или нет.
5. Нет. События подключения не инициируются при повторном рендеринге для оптимизации быстродействия, потому что операции подключения являются относительно затратными.

6 Обработка событий в React

Посмотрите вступительный видеоролик к этой главе, отсканировав QR-код или перейдя по ссылке http://reactquickly.co/videos/ch06.

ЭТА ГЛАВА ОХВАТЫВАЕТ СЛЕДУЮЩИЕ ТЕМЫ:

- Работа с событиями DOM в React.
- Реакция на события DOM, не поддерживаемые React.
- Интеграция React с другими библиотеками: UI-события jQuery.

К настоящему моменту вы узнали, как строить пользовательские интерфейсы без какого-либо взаимодействия с пользователем. Другими словами, вы только выводили данные. Для примера мы построили часы, которые не позволяли вводить данные — например, назначить часовой пояс.

На практике статические пользовательские интерфейсы встречаются редко; приходится строить элементы, достаточно умные для ответов на действия пользователей. Как отреагировать на действия пользователя — щелчки, перетаскивания указателя мыши?

Эта глава показывает, как организуется обработка событий в React. Затем, в главе 7, ваши знания о событиях будут применены к работе с веб-формами и их элементами. Я упоминал, что React поддерживает не все события; в этой главе я покажу, как работать с событиями, не поддерживаемыми React.

ПРИМЕЧАНИЕ Исходный код примеров этой главы доступен по адресам www.manning.com/books/react-quickly и https://github.com/azat-co/react-quickly/tree/master/ch06 (в папке ch06 репозитория GitHub https://github.com/azat-co/react-quickly). Также некоторые демонстрационные примеры доступны по адресу http://reactquickly.co/demos.

6.1. Работа с событиями DOM в React

А теперь посмотрим, как заставить элементы React реагировать на действия пользователя, определяя обработчики событий для таких действий. Для этого обработчик события (определение функции) определяется как значение атрибута элемента в JSX и как свойство элемента в простом коде JavaScript (когда `createElement()` вызывается напрямую без JSX). Для атрибутов, которые являются именами событий, используются стандартные имена событий W3C DOM в верблюжьем регистре, такие как `onClick` или `onMouseOver`, как в

```
onClick={function() {...}}
```

или

```
onClick={() => {...}}
```

Например, в React можно определить слушателя события, который срабатывает, когда пользователь щелкает на кнопке. В слушателе события в консоль выводится контекст `this`. Объект события представляет собой расширенную версию объекта события DOM (с именем `SyntheticEvent`):

```
<button onClick={(function(event) {
    console.log(this, event)
  }).bind(this)}>
  Save
</button>
```

Вызов `bind()` необходим, чтобы в функции — обработчике события можно было получить ссылку на экземпляр класса (элемент React). Без вызова `bind` она будет равна `null` (`use strict mode`). Связывание контекста с классом вызовом `bind(this)` не является необходимым в следующих случаях:

- Когда вам не нужно ссылаться на текущий класс с использованием `this`.
- Когда вы используете более старый стиль `React.createClass()` вместо нового стиля ES6+, потому что `createClass()` автоматически выполняет связывание за вас.
- При использовании синтаксиса «толстой стрелки» (`()=>{}`).

Также можно добиться более элегантной записи, используя метод класса в качестве обработчика события (назовем его `handleSave()`) для события `onClick`. Рассмотрим компонент `SaveButton`, который по щелчку выводит значения `this` и `event`, но использует метод класса, как показано на рис. 6.1 и в листинге 6.1 (ch06/button/jsx/button.jsx).

Листинг 6.1. Объявление обработчика события как метода класса

```
class SaveButton extends React.Component {
  handleSave(event) {
    console.log(this, event)
```

```
  }
  render() {
    return <button onClick={this.handleSave.bind(this)}>
      Save
    </button>
  }
}
```

Передает определение функции, возвращаемое bind(), событию onClick

Кнопка будет выводить в консоль значения `this` и `event`.

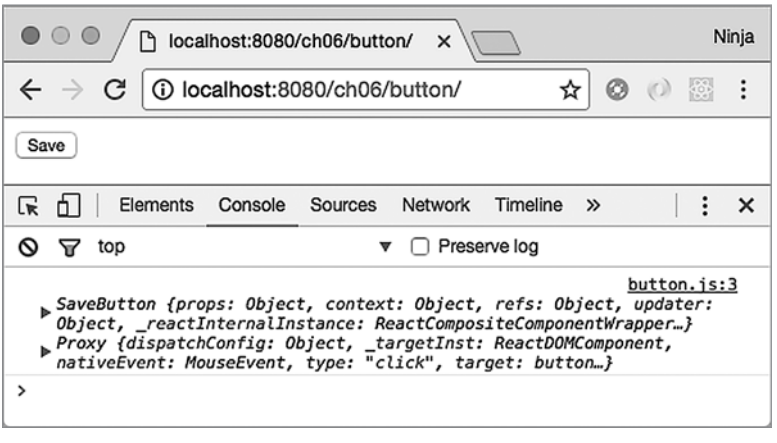


Рис. 6.1. По щелчку на кнопке выводится значение this: SaveButton

Кроме того, обработчик события можно связать с классом в конструкторе класса. С функциональной точки зрения различий нет; но если вы используете тот же метод более одного раза в `render()`, связывание в конструкторе позволит устранить дублирование кода. Та же кнопка, но со связыванием обработчика события в конструкторе:

```
class SaveButton extends React.Component {
  constructor(props) {
    super(props)
    this.handleSave = this.handleSave.bind(this)
  }
  handleSave(event) {
    console.log(this, event)
  }
  render() {
    return <button onClick={this.handleSave}>
      Save
    </button>
  }
}
```

Связывает контекст «this» с классом, чтобы использовать «this» в обработчике события для ссылок на этот класс

Передает определение функции onClick

Связывание обработчиков событий — мой любимый и рекомендуемый подход, потому что он устраняет дублирование и группирует все связывание в одном месте.

В табл. 6.1 перечислены типы событий, поддерживаемые в настоящее время в React v15. Обратите внимание на использование «верблюжьего регистра» в именах событий, чтобы они не отличались от других имен атрибутов в React.

Таблица 6.1. События DOM, поддерживаемые в React v15

Группа событий	События, поддерживаемые React
События мыши	`onClick`, `onContextMenu`, `onDoubleClick`, `onDrag`, `onDragEnd`, `onDragEnter`, `onDragExit`, `onDragLeave`, `onDragOver`, `onDragStart`, `onDrop`, `onMouseDown`, `onMouseEnter`, `onMouseLeave`, `onMouseMove`, `onMouseOut`, `onMouseOver`, `onMouseUp`
События клавиатуры	`onKeyDown`, `onKeyPress`, `onKeyUp`
События буфера обмена	`onCopy`, `onCut`, `onPaste`
События форм	`onChange`, `onInput`, `onSubmit`
События фокуса	`onFocus`, `onBlur`
События касания	`onTouchCancel`, `onTouchEnd`, `onTouchMove`, `onTouchStart`
События UI	`onScroll`
События колеса	`onWheel`
События выделения	`onSelect`
События изображений	`onLoad`, `onError`
События анимации	`onAnimationStart`, `onAnimationEnd`, `onAnimationIteration`
События переходов	`onTransitionEnd`

Как видите, React поддерживает несколько типов нормализованных событий. Если сравнить это со списком стандартных событий по адресу https://developer.mozilla.org/en-US/docs/Web/Events, вы увидите, что поддержка событий в React весьма обширна — и вы можете быть уверены в том, что команда React добавит новые события в будущем! За дополнительной информацией и именами событий обращайтесь на страницу документации по адресу http://facebook.github.io/react/docs/events.html.

6.1.1. Фазы спуска и подъема

Как вы уже знаете, React использует декларативный, а не императивный стиль программирования, что избавляет от необходимости в манипуляциях с объектами, а события присоединяются в коде не так, как это делается с jQuery (например, `$('.btn').click(handleSave)`). Вместо этого событие объявляется в JSX как атрибут

(например, `onClick={handleSave}`). Если вы объявляете события мыши, имя атрибута может быть любым из поддерживаемых событий, перечисленных в табл. 6.1. Значением атрибута является обработчик события.

Например, если вы хотите определить событие наведения указателя мыши, используйте событие `onMouseOver`, как показано в следующем примере. При наведении указателя мыши на красную рамку `<div>` в консоли Firebug или DevTools будет выводиться сообщение «mouse is over»:

```
<div
  style={{border: '1px solid red'}}
  onMouseOver={()=>{console.log('mouse is over')}} >
    Open DevTools and move your mouse cursor over here
</div>
```

События, показанные выше, например `onMouseOver`, инициируются событием в фазе *подъема* (bubble up). Также существует фаза *спуска* (trickle down/capture), предшествующая фазам подъема и цели. Сначала идет фаза спуска, от окна до целевого элемента, затем идет фаза цели, и только потом идет фаза подъема, когда событие перемещается вверх по дереву обратно к окну, как показано на рис. 6.2.

Различия между фазами особенно важны, когда одно событие используется для элемента и его предка(-ов). В режиме подъема событие сначала перехватывается, затем обрабатывается внутренним элементом (целью), а затем распространяется во внешние элементы (предки, начиная с родителя цели). В режиме спуска событие сначала перехватывается внешним элементом, а затем распространяется на внутренние элементы.

Чтобы зарегистрировать слушателя события для фазы спуска, присоедините `Capture` к имени события. Например, вместо `onMouseOver` для обработки события `mouseover` в фазе спуска используется `onMouseOverCapture`. Это относится ко всем именам событий, перечисленным в табл. 6.1.

Допустим, имеется элемент `<div>` с обычным событием (подъема) и спускающимся событием. Эти события определяются с именами `onMouseOver` и `onMouseOverCapture` соответственно (ch06/mouse-capture/jsx/mouse.jsx).

Листинг 6.2. Спускающееся событие, за которым следует поднимающееся событие

```
class Mouse extends React.Component {
  render() {
    return <div>
      <div
        style={{border: '1px solid red'}}
        onMouseOverCapture={((event)=>{
          console.log('mouse over on capture event')
          console.dir(event, this)}).bind(this)}
        onMouseOver={((event)=>{
          console.log('mouse over on bubbling event')
          console.dir(event, this)}).bind(this)} >
```

```
          Open DevTools and move your mouse cursor over here
        </div>
      </div>
    }
}
```

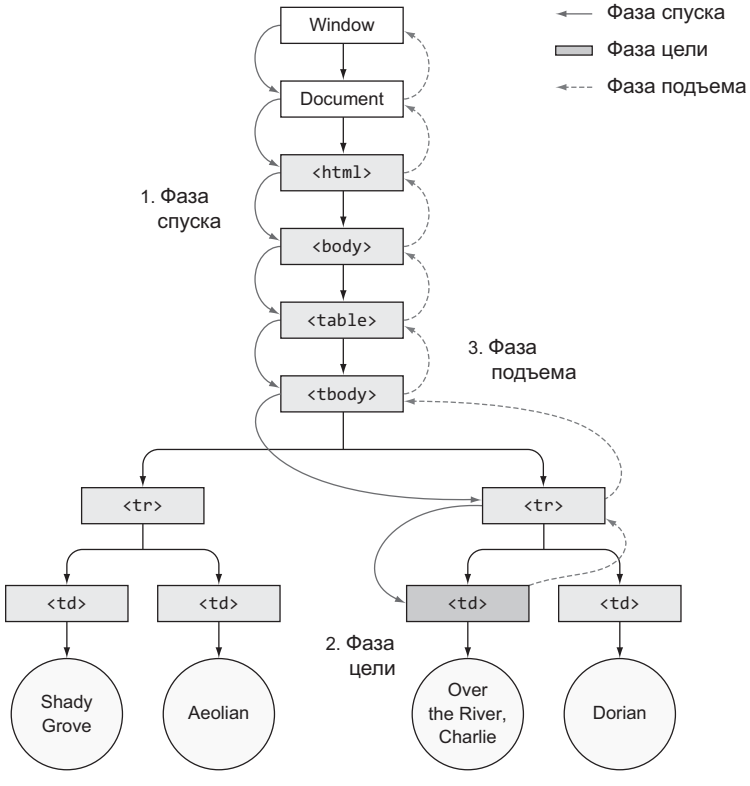


Рис. 6.2. Фазы спуска, цели и подъема

Контейнер имеет красную рамку толщиной в 1 пиксел; он содержит текст, показанный на рис. 6.3, чтобы вы знали, куда навести курсор. Каждое событие `mouseover` выводит как тип события, так и объект события (скрытый в разделе `Proxy` в DevTools на рис. 6.3 из-за использования `console.dir()`).

Понятно, что спускающееся событие (`capture`) регистрируется первым. Это поведение может использоваться для остановки распространения и назначения приоритетов между событиями.

Очень важно понимать, как в React реализованы события, потому что события являются краеугольным камнем пользовательских интерфейсов. В главе 7 события React рассматриваются более подробно.

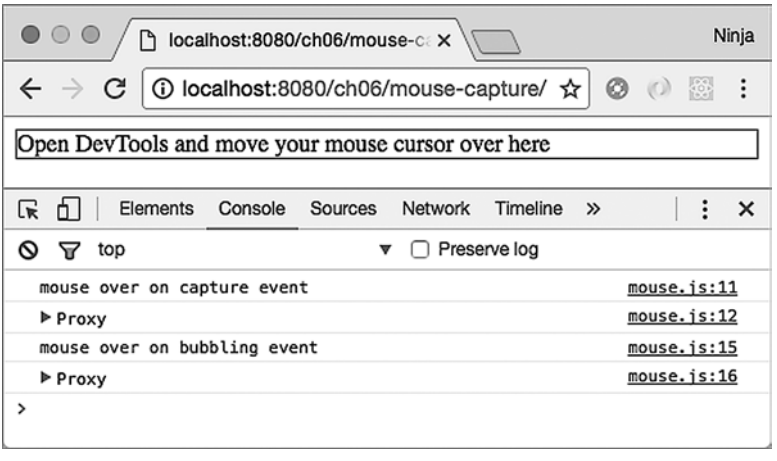


Рис. 6.3. Спускающееся событие происходит перед обычным событием

6.1.2. События React во внутренней реализации

События в React работают не так, как в jQuery или простом коде JavaScript, которые обычно помещают слушатель события прямо в узел DOM. Когда вы помещаете события прямо в узлы, могут возникнуть проблемы с удалением и добавлением событий в ходе жизненного цикла пользовательского интерфейса. Допустим, у вас имеется список учетных записей, каждую из которых можно удалить или отредактировать; также в список можно добавить новые учетные записи. Разметка HTML может выглядеть примерно так (каждый элемент учетной записи `<li>` однозначно определяется значением `id`):

```
<ul id="account-list">
    <li id="account-1">Account #1</li>
    <li id="account-2">Account #2</li>
    <li id="account-3">Account #3</li>
    <li id="account-4">Account #4</li>
    <li id="account-5">Account #5</li>
    <li id="account-6">Account #6</li>
</ul>
```

Если учетные записи часто удаляются или добавляются в список, управление записями существенно усложняется. Было бы лучше, чтобы один слушатель события у родителя (`account-list`) прослушивал поднимающиеся события (событие поднимается вверх по дереву DOM, если ничего не перехватит его на более низком уровне). Во внутренней реализации React отслеживает события, присоединенные к элементам более высоких уровней и целевым элементам, в отображении, или маппинге (mapping). Это позволяет React отследить цель от родителя (`document`), как показано на рис. 6.4.

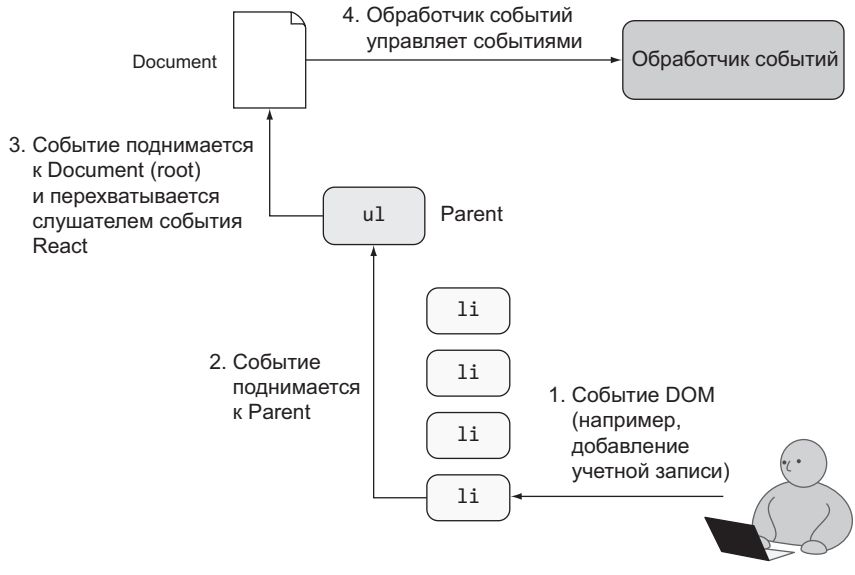


Рис. 6.4. Событие DOM (1) поднимается к своим предкам (2-3), где оно перехватывается обычным (стадия подъема) слушателем события React (4), потому что в React события перехватываются в root (Document)

Посмотрим, как работает делегирование события родителю, на примере компонента `Mouse` из листинга 6.2. Имеется элемент `<div>` с событием React `mouseover`; задача — проанализировать события для этого элемента.

Если открыть Chrome DevTools или Firefox Tools и выбрать элемент `data-reactroot` на вкладке Elements или Inspector (или выбрать команду Inspect в контекстном меню Chrome или команду Inspect Element в контекстном меню Firefox), вы сможете обратиться к `<div>` из консоли (другая вкладка в DevTools/Firebug) — введите `$0` и нажмите Enter (удобный трюк).

Интересно, что узел DOM `<div>` не содержит слушателей событий. `$0` — элемент `<div>` и `reactroot` (рис. 6.5). Следовательно, вы можете проверить, какие события присоединены к этому конкретному элементу (узлу DOM), вызовом глобального метода `getEventListeners()` из консоли DevTools:

```
getEventListeners($0)
```

Результат — пустой объект `{}`. React не присоединяет слушателей событий к узлу `reactroot <div>`. Но если навести указатель мыши на элемент, информация выводится в консоль — очевидно, событие было перехвачено! Куда же оно делось?

Попробуйте повторить процедуру с `<div id="content">` или с элементом `<div>` с красной рамкой (дочерним элементом `reactroot`). Для каждого элемента, выбранного на вкладке Elements, `$0` будет обозначать выбранный элемент, поэтому выберите новый элемент и повторите команду `getEventListeners($0)`. По-прежнему ничего?

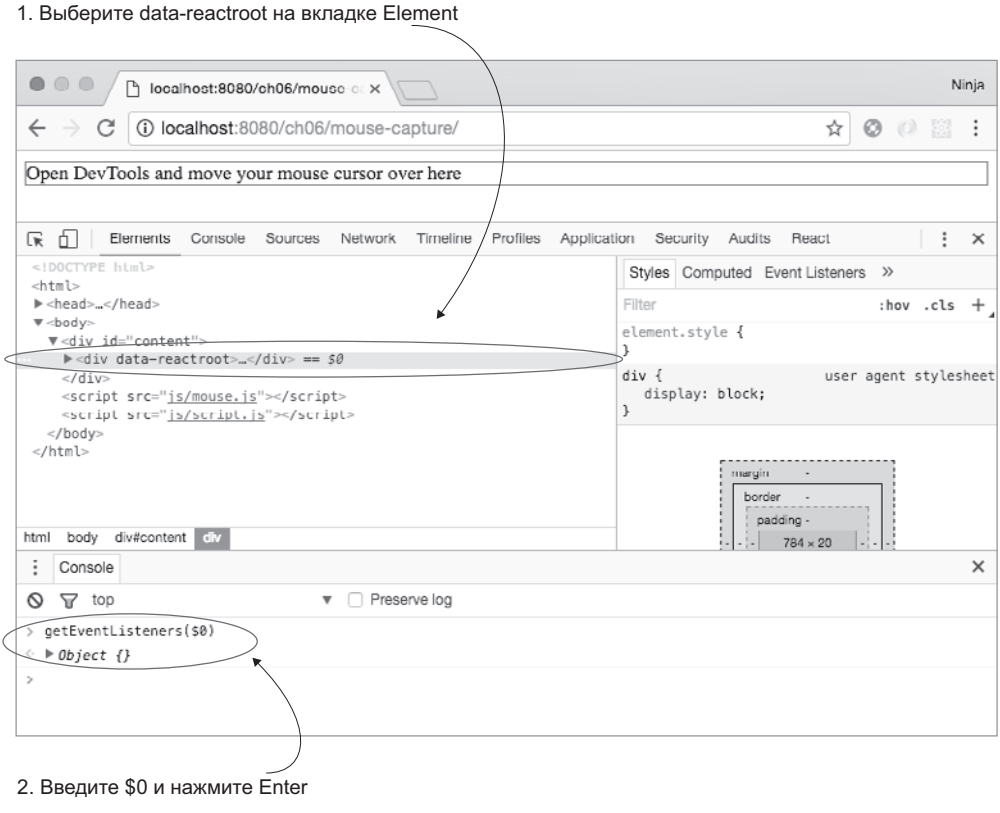


Рис. 6.5. Анализ событий элемента (ничего не найдено)

Хорошо. Проанализируем события `document`, выполнив следующий код в консоли:

```
getEventListeners(document)
```

Бум! На этот раз событие обнаруживается: `Object {mouseover: Array[1]}`, как показано на рис. 6.6. Теперь вы знаете, что React присоединяет слушателя события к *главному* предку всех элементов — элементу `document`. Событие не было присоединено ни к отдельному узлу (такому, как `<div>`), ни к элементу с атрибутом `data-reactroot`.

Событие можно удалить выполнением следующей команды в консоли:

```
getEventListeners(document).mouseover[0].remove()
```

Теперь событие не будет появляться при перемещении указателя мыши. Слушатель события, присоединенный к документу, исчез; это доказывает, что React присоединяет события к `document`, а не к каждому элементу. Это позволяет React работать

быстрее, особенно со списками. Данный подход отличается от принципа работы jQuery: в этой библиотеке события присоединяются к отдельным элементам. Спасибо разработчикам React за то, что подумали о быстродействии.

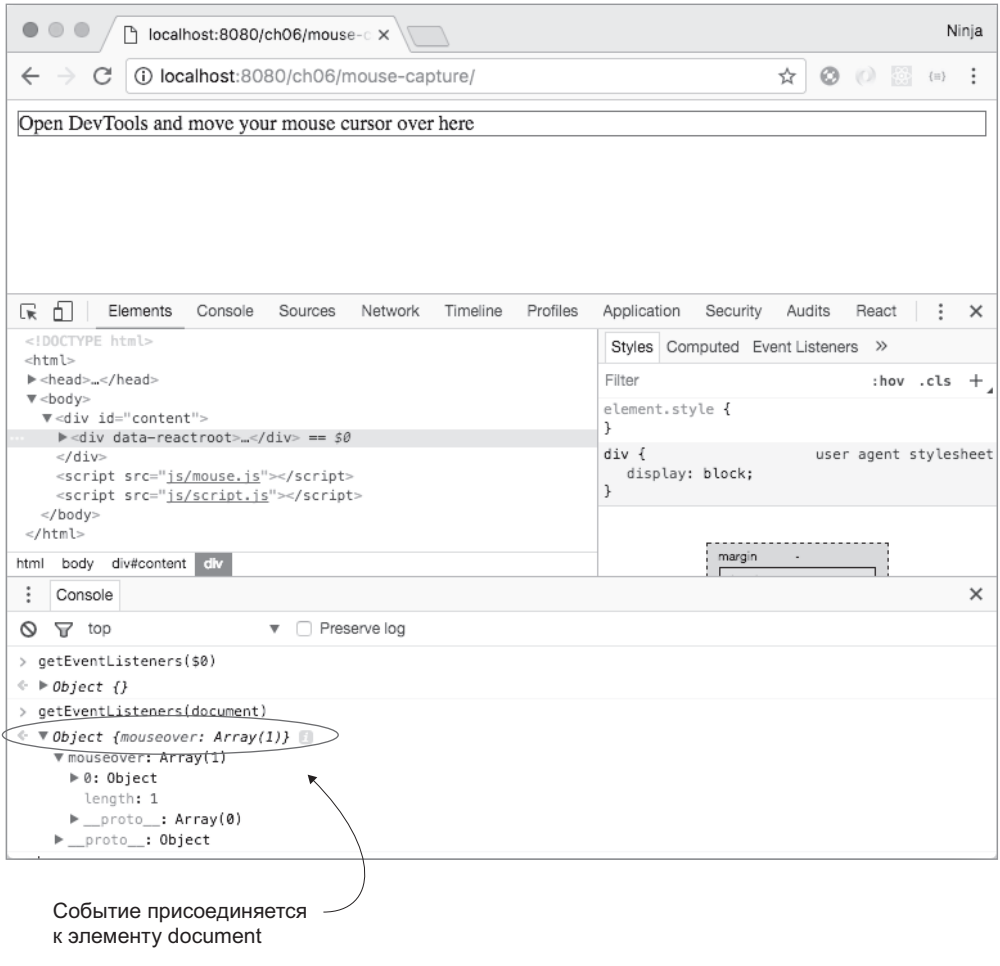


Рис. 6.6. Анализ событий в элементе document (на этот раз событие обнаружено)

Если у вас имеются другие элементы с тем же типом события (например, два события `mouseover`), они связываются с одним событием, а внутренний маппинг React связывает событие с правильным дочерним (целевым) элементом, как показано на рис. 6.7. А если говорить о целевых элементах, вы можете получить информацию о целевом узле (в котором произошло событие) из объекта события.

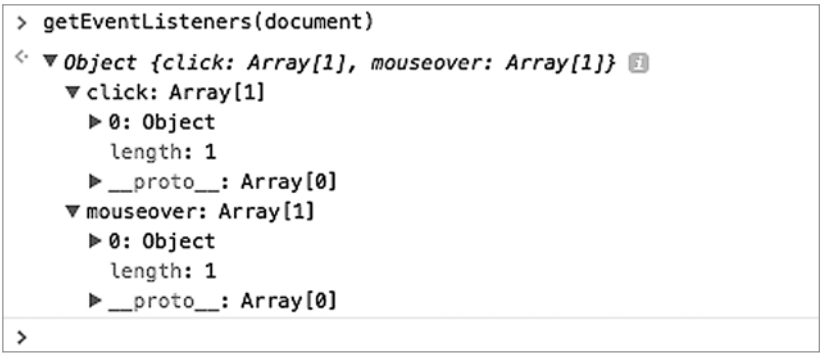


Рис. 6.7. React повторно использует слушателей события в корне, поэтому вы увидите только одного слушателя даже при существовании нескольких элементов с mouseover

6.1.3. Работа с объектами события React SyntheticEvent

Браузеры могут отличаться по своим реализациям спецификации W3C (см. www.w3.org/TR/DOM-Level-3-Events). Когда вы работаете с событиями DOM, объект события, передаваемый обработчику, может обладать разными свойствами и методами. Это может вызвать межбраузерные проблемы при написании кода обработки событий. Например, для получения целевого элемента в IE версии 8 вам пришлось бы обратиться к `event.srcElement`, тогда как в Chrome, Safari и Firefox используется `event.target`:

```
var target = event.target || event.srcElement
console.log(target.value)
```

Конечно, в 2016 году ситуация с межбраузерной совместимостью гораздо лучше, чем в 2006 году. И все же хочется ли вам тратить время на чтение спецификаций и отладку из-за неочевидных различий между браузерными реализациями? Мне не хочется.

Проблемы межбраузерной совместимости неприятны, потому что пользователи должны получать одинаковый опыт взаимодействия в разных браузерах. Обычно приходится добавлять лишний код (например, команды `if/else`) для компенсации различий браузерных API. Вам также придется провести дополнительное тестирование в разных браузерах. Короче говоря, поиск обходных решений и исправление проблем межбраузерной совместимости создает больше неприятностей, чем проблемы CSS, проблемы IE8 или придирчивые дизайнеры в хипстерских очках.

У React есть решение: обертка для «родных» событий браузеров. С оберткой события соответствуют спецификации W3C независимо от того, в каком браузере вы открываете свои страницы. Во внутренней реализации React использует соб-

ственный класс для *синтетических событий* (`SyntheticEvent`). Экземпляры класса `SyntheticEvent` передаются обработчику событий. Например, чтобы получить доступ к объекту синтетического события, вы добавляете аргумент `event` функции-обработчику события, как показано в листинге 6.3 (ch06/mouse/jsx/mouse.jsx). Объект события выводится в консоль (рис. 6.8).

Листинг 6.3. Получение синтетического события обработчиком

```
class Mouse extends React.Component {
  render() {
    return <div>
      <div
        style={{border: '1px solid red'}}
        onMouseOver={((event)=>{
          console.log('mouse is over with event')
          console.dir(event)})} >
          Open DevTools and move your mouse cursor over here
        </div>
    </div>
  }
}
```

Определяет аргумент event

Обращается к объекту SyntheticEvent для интерактивного вывода (dir)

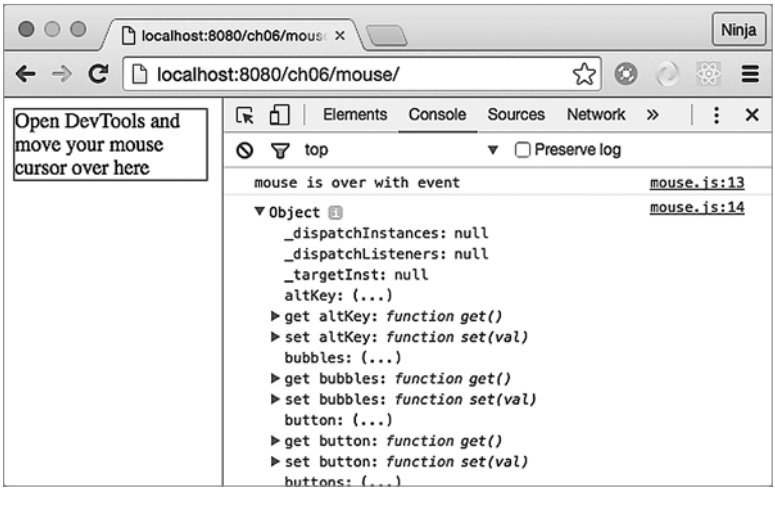


Рис. 6.8. При наведении указателя мыши на поле объект события выводится в консоль DevTools

Как вы уже видели, код обработки события можно переместить в метод компонента или в отдельную функцию. Например, можно создать метод `handleMouseOver()` с использованием синтаксиса метода класса ES6+/ES2015+ и сослаться на него из возвращаемого значения `render()` в форме `{this.handleMouseOver.bind(this)}`. Вызов `bind()` необходим для передачи правильного значения `this` в функцию.

При использовании синтаксиса `=>`, как было сделано в предыдущем примере, это происходит автоматически. Также это происходит автоматически с синтаксисом `createClass()`. Впрочем, с этим классом этого не происходит. Конечно, если `this` в методе не используется, связывание не потребуется; просто используйте `onMouseOver={this.handleMouseOver}`.

Имя `handleMouseOver()` выбрано произвольно (в отличие от имен событий жизненного цикла, рассмотренных в главе 5) и не обязано подчиняться никаким соглашениям — при условии, что оно понятно вам и вашей команде. Как правило, в React обработчик события снабжается префиксом `handle`, чтобы отличить его от обычных методов класса, и именем события (например, `mouseOver`) или именем операции (например, `save`).

Листинг 6.4. Обработчик события как метод класса: связывание в render()

```
class Mouse extends React.Component {
  handleMouseOver(event) {
    console.log('mouse is over with event')
    console.dir(event.target)
  }
  render(){
    return <div>
      <div
        style={{border: '1px solid red'}}
        onMouseOver={this.handleMouseOver.bind(this)} >
          Open DevTools and move your mouse cursor over here
        </div>
    </div>
  }
}
```

Событие обладает теми же свойствами или методами, что и большинство «родных» браузерных событий, в том числе `stopPropagation()`, `preventDefault()`, `target` и `currentTarget`. Если вы не можете найти «родное» свойство или метод, вы можете обратиться к «родному» событию браузера при помощи `nativeEvent`:

```
event.nativeEvent
```

Ниже перечислены некоторые атрибуты и методы интерфейса React v15.x `SyntheticEvent`:

- `currentTarget` –– `DOMEventTarget` элемента, перехватившего событие (может быть целью или родителем цели).
- `target` — `DOMEventTarget`, то есть элемент, в котором было инициировано событие.
- `nativeEvent` — `DOMEvent`, «родной» объект события браузера.
- `preventDefault()` — блокирует поведение по умолчанию (например, ссылку или кнопку отправки данных формы).

- `isDefaultPrevented()` — логическое значение: `true`, если поведение по умолчанию было заблокировано.
- `stopPropagation()` — останавливает распространение события.
- `isPropagationStopped()` — логическое значение: `true`, если распространение было заблокировано.
- `type` — строковое имя.
- `persist()` — удаляет синтетическое событие из пула и разрешает сохранение ссылок на событие в пользовательском коде.
- `isPersistent` — логическое значение: `true`, если объект `SyntheticEvent` был удален из пула.

Упомянутое свойство `target` объекта события содержит узел DOM объекта, с которым произошло событие, а не того, где оно было перехвачено, как в случае с `currentTarget` (https://developer.mozilla.org/en-US/docs/Web/API/Event/target). Чаще всего при построении пользовательских интерфейсов также требуется получить текст поля ввода. Его можно получить из `event.target.value`.

Синтетическое событие *обнуляется* (то есть становится недействительным) после того, как обработчик события завершит работу. Вы можете сохранить ссылку на то же событие в переменной, чтобы обратиться к нему позднее или асинхронно (в будущем) в функции обратного вызова. Например, можно сохранить ссылку на объект события в глобальной переменной `e` (ch06/mouseevent/jsx/mouse.jsx).

Листинг 6.5. Обнуление синтетического события

```
class Mouse extends React.Component {
  handleMouseOver(event) {
    console.log('mouse is over with event')
    window.e = event // Антипаттерн
    console.dir(event.target)  <-- Использует объект события и его атрибуты в методе
    setTimeout(()=>{
      console.table(event.target)
      console.table(window.e.target)  <-- По умолчанию событие не может использоваться в асинхронном обратном вызове или вызовом window.e
    }, 2345)
  }
  render() {
    return <div>
      <div
        style={{border: '1px solid red'}}
        onMouseOver={this.handleMouseOver.bind(this)}>
          Open DevTools and move your mouse cursor over here
        </div>
    </div>
  }
}
```

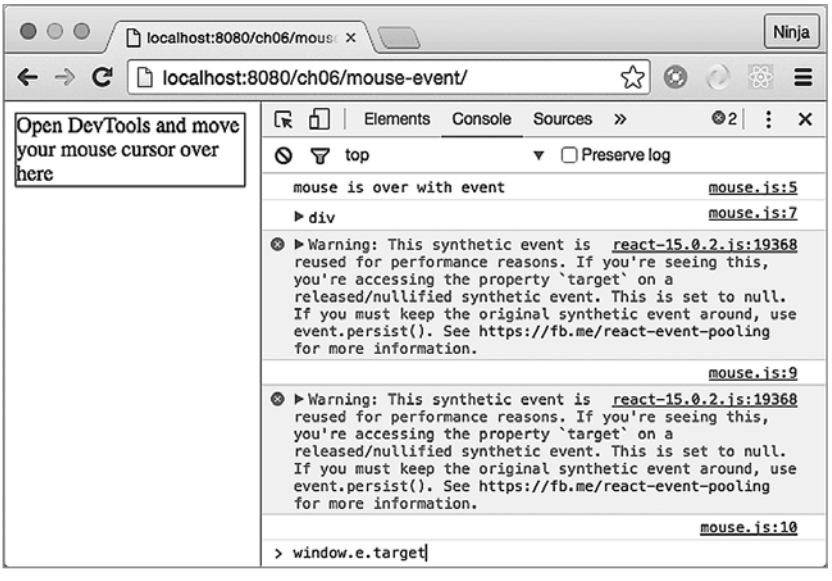


Рис. 6.9. Сохранение объекта синтетического события для использования в будущем невозможно по умолчанию — отсюда и предупреждение

Вы получите предупреждение о том, что React повторно использует синтетическое событие по соображениям быстродействия (рис. 6.9):

```
Это синтетическое событие используется повторно по соображениям быстродействия.
Если вы видите
➥ это предупреждение, вы обращаетесь к свойству `target` освобожденного/
  обнуленного
➥ синтетического события. Ему присвоено значение null.
```

Если вам нужно хранить синтетическое событие после того, как обработчик события будет выполнен, используйте метод `event.persist()`. Когда вы применяете его, объект события не будет повторно использован и обнулен.

Вы видели, что React даже синтезирует (или нормализует) событие браузера за вас; это означает, что React создает межбраузерную обертку для «родных» объектов событий. Преимущество такого решения заключается в том, что события работают одинаково практически во всех браузерах. И в большинстве случаев для события React будут доступны все «родные» методы, включая `event.stopPropagation()` и `event.preventDefault()`. Но если вам все равно понадобится обратиться к «родному» событию, оно хранится в свойстве `event.nativeEvent` объекта синтетического события. Разумеется, если вы работаете с «родными» событиями напрямую, вам нужно знать обо всех различиях межбраузерной совместимости и уметь работать с ними.

6.1.4. Использование событий и состояния

Использование состояний с событиями, или, другими словами, возможность изменить состояние компонента в ответ на событие, позволит вам создавать интерактивные пользовательские интерфейсы, реагирующие на действия пользователя. Это весьма интересная возможность, потому что вы сможете перехватывать любые события и изменять представления на основании этих событий и логики приложения. При этом компоненты становятся автономными, потому что они не нуждаются во внешнем коде или представлении.

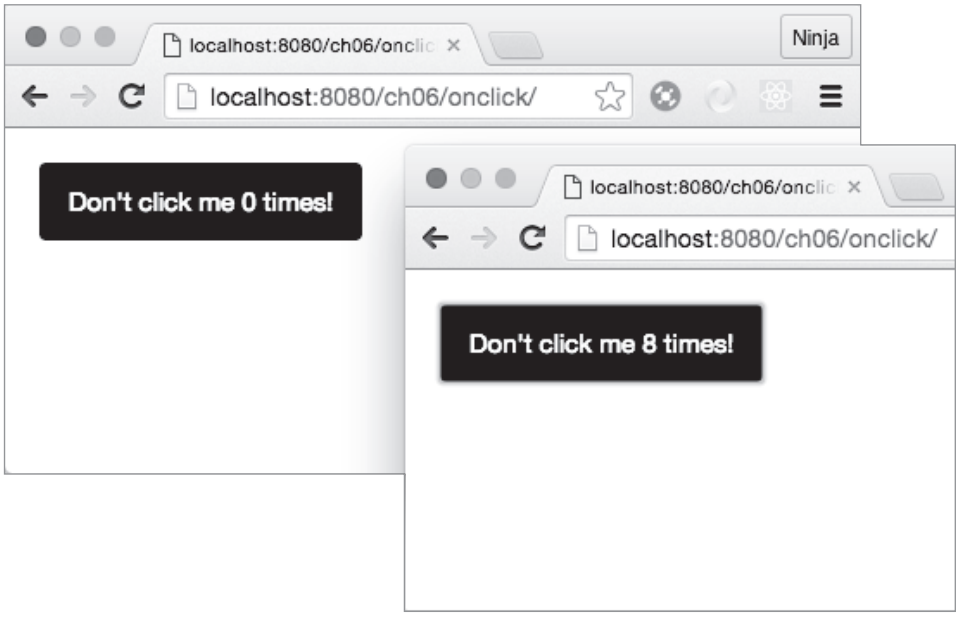


Рис. 6.10. Щелчок на кнопке увеличивает счетчик, исходное значение которого равно 0

Например, реализуем кнопку с текстом, с которой связан счетчик. Исходное значение счетчика равно 0, как показано на рис. 6.10. Каждый щелчок на кнопке увеличивает число, выводимое на кнопке (1, 2, 3 и т. д.).

Работа начинается с реализации следующих методов:

- `constructor()` — значение `this.state` равно 1, потому что счетчику необходимо присвоить 0 перед его использованием в представлении.
- `handleClick()` — обработчик события, увеличивающий счетчик.
- `render()` — метод рендера, возвращающий кнопку JSX.

Метод `click()` имеет много общего с другими методами компонентов React. Помните `getUrl()` из главы 3 и `handleMouseOver()` ранее в этой главе? Этот метод компонента объявляется аналогично, если не считать того, что контекст придется связать вручную. Метод `handleClick()` присваивает состоянию `counter` текущее значение `counter`, увеличенное на 1 (ch06/onclick/jsx/content.jsx).

Листинг 6.6. Обновление состояния в результате действия щелчка

```
class Content extends React.Component {
  constructor(props) {
    super(props)
    this.state = {counter: 0}
  }
  handleClick(event) {
    this.setState({counter: ++this.state.counter})
  }
  render() {
    return (
      <div>
        <button
          onClick={this.handleClick.bind(this)}
          className="btn btn-primary">
          Don't click me {this.state.counter} times!
        </button>
      </div>
    )
  }
}
```

Присваивает 0 счетчику в исходном состоянии

Увеличивает значение счетчика на 1

Присоединяет слушателя события onClick к триггеру handleClick

Выводит значение счетчика из состояния

ВЫЗОВ И ОПРЕДЕЛЕНИЕ

Просто для напоминания: вы заметили, что хотя `this.handleClick()` является методом в листинге 6.6, он не вызывается в JSX при присваивании `onClick` (то есть `<button onClick={this.handleClick})`?). Иначе говоря, после `this.handleClick()` в фигурных скобках нет круглых скобок (`()`). Дело в том, что вам нужно передать определение функции, а не вызывать ее. Функции являются полноправными объектами в JavaScript, а в данном случае определение функции передается как значение атрибута `onClick`. С другой стороны, метод `bind()` вызывается, потому что он позволяет использовать правильное значение `this`, но `bind()` возвращает определение функции. Таким образом, вы все равно получаете определение функции как значение `onClick`.

Помните, как упоминалось выше, что `onClick` не является реальным атрибутом HTML, хотя с точки зрения синтаксиса он не отличается от любого другого объявления JSX (например, `className={btnClassName}` или `href={this.props.url}`).

Если щелкать на кнопке, счетчик будет увеличиваться с каждым щелчком. На рис. 6.10 показано, что я щелкнул на кнопке 8 раз: сейчас счетчик равен 8, а изначально он был равен 0. Гениально, не правда ли?

Как и в случае с `onClick` или `onMouseOver`, вы можете использовать любые события DOM, поддерживаемые React. Фактически вы определяете представление и обработчик события, который изменяет состояние. Вы не пишете императивный код для изменения представления. Такова сила декларативного стиля!

В следующем разделе вы узнаете, как передавать обработчики событий и другие объекты дочерним элементам.

6.1.5. Передача обработчиков событий как свойств

Представьте следующую ситуацию: имеется кнопка, которая является компонентом без состояния. Все, что у нее есть, — стилевое оформление. Как присоединить слушателя события, чтобы кнопка могла инициировать выполнение некоторого кода?

Давайте на минутку вернемся к свойствам. Как вы уже знаете, свойства неизменяемы. Они передаются родительскими компонентами своим дочерним элементам. Так как функции являются полноправными объектами в JavaScript, вы можете создать в дочернем элементе свойство, которое является функцией, и использовать его в качестве обработчика события.

Решение проблемы, упоминавшейся ранее, — инициирования события из компонента без состояния, — заключается в передаче обработчика события как свойства этому компоненту без состояния и использования свойства (функции обработчика события) в компоненте без состояния (вызова функции). Например, разобьем функциональность предыдущего примера на два компонента: `ClickCounterButton` и `Content`. Первый является компонентом без состояния («глупый компонент»), а второй обладает состоянием («умный компонент»).

ПРЕЗЕНТАЦИОННЫЕ И КОНТЕЙНЕРНЫЕ КОМПОНЕНТЫ

Глупые и умные компоненты иногда называются «презентационными» и «контейнерными» компонентами соответственно. Эта дихотомия связана с отсутствием или наличием состояния, но не всегда точно совпадает с ним.

В большинстве случаев презентационные компоненты не обладают состоянием и могут быть компонентами без состояния (или функциональными компонентами). Это не всегда так, потому что вам может потребоваться состояние, относящееся к презентационному уровню.

Презентационные/глупые компоненты часто используют `this.props.children` и выполняют рендер элементов DOM. С другой стороны, контейнерные/умные компоненты описывают то, как все работает, без элементов DOM, обладают состоянием, обычно используют паттерны компонентов более высокого порядка и подключаются к источникам данных.

На практике рекомендуется использовать комбинацию глупых и умных компонентов. Это позволяет сохранить чистоту структуры и обеспечивает лучшее разделение обязанностей.

При выполнении кода значение счетчика увеличивается с каждым щелчком. На визуальном уровне ничего не изменилось по сравнению с предыдущим примером с кнопкой и счетчиком (рис. 6.10), однако во внутренней реализации появился компонент `ClickCounterButton` (без состояния и в общем-то без логики) в дополнение к `Content`, который все еще содержит всю логику.

Компонент `ClickCounterButton` не имеет собственного обработчика события `onClick` (то есть `this.handler` или `this.handleClick`). Он использует обработчик, переданный ему родителем в свойстве `this.props.handler`. В общем случае использование этого механизма полезно для обработки событий кнопки, потому что кнопка является презентационным/глупым компонентом без состояния. Эту кнопку можно повторно использовать в других пользовательских интерфейсах.

В листинге 6.7 показан код презентационного компонента, который рендерит кнопку (ch06/onclick-props/jsx/click-counter-button.jsx); родитель `Content`, который рендерит этот элемент, приведен в листинге 6.8.

Листинг 6.7. Компонент кнопки без состояния

```
class ClickCounterButton extends React.Component {
  render() {
    return <button
      onClick={this.props.handler}
      className="btn btn-danger">
      Increase Volume (Current volume is {this.props.counter})
    </button>
  }
}
```

Компонент `ClickCounterButton` на рис. 6.11 предельно тривиален, но этим-то и хороша данная архитектура. Компонент получается очень простым и понятным.

Компонент `ClickCounterButton` также использует свойство `counter`, которое рендерится конструкцией `{this.props.counter}`. Передача свойств дочерним элементам (таким, как `ClickCounterButton`) осуществляется просто, если вы еще не забыли примеры из главы 2. Для этого используется стандартный синтаксис атрибутов: `name=VALUE`.

Например, чтобы передать свойства `counter` и `handler` компоненту `ClickCounterButton`, укажите атрибуты в объявлении JSX параметра `render` родителя (в данном случае родителем является `Content`):

```
<div>
  <ClickCounterButton
    counter={this.state.counter}
    handler={this.handleClick}/>
</div>
```

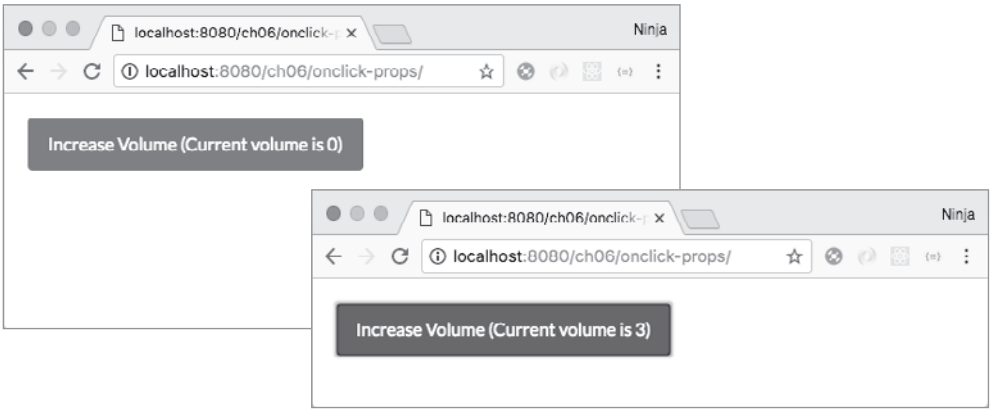


Рис. 6.11. Передача обработчика события в свойстве кнопке (презентационный компонент) позволяет увеличивать счетчик в надписи на кнопке, который также является свойством кнопки

`counter` в `ClickCounterButton` является свойством, а следовательно, *не может* изменяться; но в родителе `Content` — это состояние, и оно *может* изменяться. (Об отличиях между свойствами и состоянием было рассказано в главе 4.) Естественно, имена могут отличаться. Вам не обязательно использовать те же имена при передаче свойств дочерним элементам. Впрочем, я считаю, что совпадение имен помогает понять, что данные совместно используются разными компонентами.

Итак, что происходит? Исходное значение `counter` (состояние) задается равным нулю в родителе `Content`. Обработчик события также определяется в родителе. Таким образом, дочерний элемент (`ClickCounterButton`) инициирует событие на стороне родителя. Код родительского компонента `Content` с `constructor()` и `handleClick()` приведен в листинге 6.8 (ch06/onclickprops/jsx/content.jsx).

Листинг 6.8. Передача обработчика события в свойстве

```
class Content extends React.Component {
  constructor(props) {
    super(props)
    this.handleClick = this.handleClick.bind(this)
    this.state = {counter: 0}
  }
  handleClick(event) {
    this.setState({counter: ++this.state.counter})
  }
  render() {
    return (
      <div>
        <ClickCounterButton
          counter={this.state.counter}
```

Связывает контекст в конструкторе, чтобы вы могли использовать метод this.setState(), относящийся к текущему экземпляру класса Content

```
          handler={this.handleClick}/>
      </div>
    )
  }
}
```

Как я говорил выше, в JavaScript функции являются полноправными объектами и их можно передавать в переменных или свойствах. Таким образом, здесь быть сюрпризов не должно. Но теперь возникает вопрос: где размещать логику таких обработчиков событий — в дочернем или родительском компоненте?

6.1.6. Передача данных между компонентами

В предыдущем примере обработчик события щелчка находился в родительском элементе. Обработчик события можно разместить и в дочернем элементе, но использование родителя позволяет обмениваться информацией между дочерними компонентами.

Используем кнопку в примере, но на этот раз удалим значение счетчика из `render()` (1, 2, 3 и т. д.). Компоненты представляют собой специализированные фрагменты представления, так что счетчик будет находиться в другом компоненте: `Counter`. А значит, всего в архитектуре будут задействованы три компонента: `ClickCounterButton`, `Content` и `Counter`.

Как видно из рис. 6.12, теперь на странице отображаются два компонента: кнопка и текст под ней. У каждого есть свойства, которые являются состояниями в родителе `Content`. В отличие от предыдущего примера (см. рис. 6.11), для подсчета щелчков здесь необходимо организовать передачу данных между кнопкой и текстом. Другими словами, `ClickCounterButton` и `Counter` должны общаться друг с другом. Для этого они используют `Content`, а *не передают* данные напрямую (прямой обмен данными нежелателен, потому что он создает сильную связь).

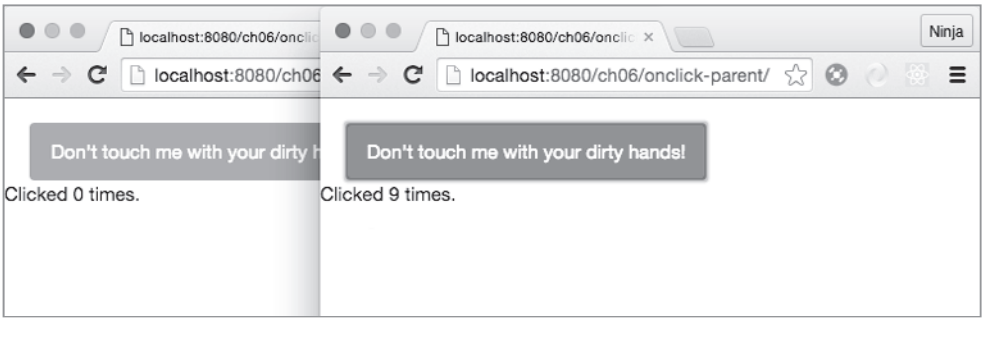


Рис. 6.12. Разделение состояния и работа с двумя дочерними компонентами без состояния (которые обмениваются информацией через родителя): для счетчика и для кнопки

Компонент `ClickCounterButton` не имеет состояния, как и в предыдущем примере. Собственно, большинство компонентов React должно быть именно таким: никаких выкрутасов, только свойства и JSX.

Листинг 6.9. Компонент кнопки использует обработчик события из Content

```
class ClickCounterButton extends React.Component {
  render() {
    return <button
      onClick={this.props.handler}
      className="btn btn-info">
      Don't touch me with your dirty hands!
    </button>
  }
}
```

Конечно, `ClickCounterButton` также можно записать в виде функции (вместо класса), чтобы немного упростить синтаксис:

```
const ClickCounterButton = (props) => {
  return <button
    onClick={props.handler}
    className="btn btn-info">
    Don't touch me with your dirty hands!
  </button>
}
```

Следующий новый компонент `Counter` выводит свойство `value`, которое используется в качестве счетчика (имена могут быть разными — вы не обязаны непременно использовать имя `counter`):

```
class Counter extends React.Component {
  render() {
    return <span>Clicked {this.props.value} times.</span>
  }
}
```

Наконец, мы добрались до родительского компонента, который предоставляет свойства: в одном передается обработчик события, а в другом счетчик. Параметр `render` необходимо изменить соответствующим образом, но остальной код остается без изменения (ch06/onclick-parent/jsx/content.jsx).

Листинг 6.10. Передача обработчика события и состояния двум компонентам

```
class Content extends React.Component {
  constructor(props) {
    super(props)
    this.handleClick = this.handleClick.bind(this)
    this.state = {counter: 0}
  }
  handleClick(event) {
```

```
    this.setState({counter: ++this.state.counter})
  }
  render() {
    return (
      <div>
        <ClickCounterButton handler={this.handleClick}/>
        <br/>
        <Counter value={this.state.counter}/>
      </div>
    )
  }
}
```

Что касается исходного вопроса о том, где следует размещать логику обработки событий, можно руководствоваться простым практическом правилом: если вам нужно взаимодействие между дочерними компонентами, размещайте ее в родителе или компоненте-обертке. Если событие относится только к дочерним компонентам, нет необходимости захламлять компоненты выше в иерархии методами обработки событий.

6.2. Реакция на события DOM, не поддерживаемые React

В табл. 6.1 перечислены события, поддерживаемые React. А как же события DOM, которые не поддерживаются React? Предположим, вам поручено создать масштабируемый пользовательский интерфейс, который должен увеличиваться или уменьшаться в зависимости от события размера окна (`resize`). Однако это событие не поддерживается! Конечно, существует способ перехвата событий `resize` и любых других событий, и вы уже знаете, какая возможность React используется для его реализации: *события жизненного цикла*.

В этом примере будут реализованы кнопки-переключатели. Как вы уже знаете, стандартные элементы кнопок-переключателей в HTML плохо и непоследовательно масштабируются (увеличиваются и уменьшаются) в разных браузерах. По этой причине во время своей работы в DocuSign я реализовал масштабируемые кнопки-переключатели CSS (http://mng.bz/kPMu) на замену стандартных переключателей HTML. Я сделал это в jQuery. Кнопки CSS могли масштабироваться через jQuery посредством манипуляции с CSS. Посмотрим, как создать масштабируемый переключатель в React. Те же переключатели CSS будут масштабироваться средствами React при изменении размера экрана, как показано на рис. 6.13.

Как упоминалось ранее, событие `resize` не поддерживается React, — попытка добавить его в элемент так, как показано в следующем листинге, не сработает:

```
...
  render() {
    return <div>
```

```
      <div onResize={this.handleResize}
        className="radio-tagger"
        style={this.state.taggerStyle}>
    ...
```

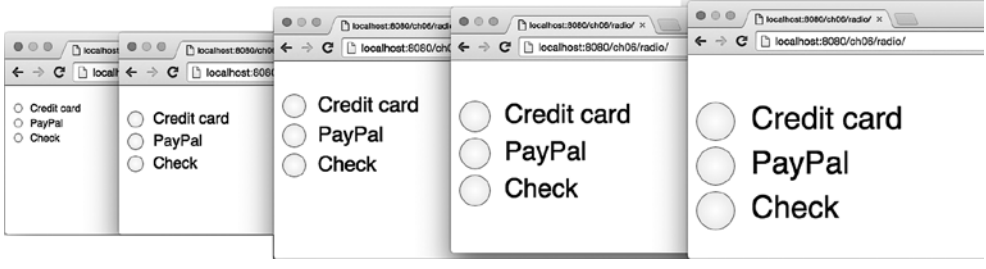


Рис. 6.13. Масштабируемые переключатели CSS под управлением React, прослушивающие событие изменения размеров окна. С изменением размеров окна уменьшаются и размеры кнопок

Существует простой способ присоединения неподдерживаемых событий типа `resize` и большинства нестандартных элементов, которые необходимо поддерживать: события жизненного цикла React. В листинге 6.11 (ch06/radio/jsx/radio.jsx) слушатели события `resize` добавляются к `window` в `componentDidMount()`, а затем те же слушатели удаляются в `componentWillUnmount()`, чтобы после удаления компонента из DOM не оставалось ничего лишнего. Слушатели событий, оставшиеся после удаления своих компонентов, — верный способ создания утечек памяти, которые могут в какой-то момент привести к аварийному завершению вашего приложения. Поверьте, утечка памяти может стать причиной многих бессонных ночей, которые вы проводите за тестированием — с воспаленными глазами, банкой Red Bull и ругательствами.

Листинг 6.11. Использование событий жизненного цикла для прослушивания событий DOM

```
class Radio extends React.Component {
  constructor(props) {
    super(props)
    this.handleResize = this.handleResize.bind(this)
    let order = props.order
    leti=1
    this.state = {   <-- Сохраняет стили в состоянии
      outerStyle: this.getStyle(4, i),
      innerStyle: this.getStyle(1, i),
      selectedStyle: this.getStyle(2, i),
      taggerStyle: {top: order*20, width: 25, height: 25}
    }
  }
```

```
getStyle(i, m) {
  let value = i*m
  return {
    top: value,
    bottom: value,
    left: value,
    right: value,
  }
}
componentDidMount() {
  window.addEventListener('resize', this.handleResize)
}
componentWillUnmount() {
  window.removeEventListener('resize', this.handleResize)
}
handleResize(event) {
  letw=1+Math.round(window.innerWidth / 300)
  this.setState({
    taggerStyle: {top: this.props.order*w*10, width: w*10, height: w*10},
    textStyle: {left: w*13, fontSize: 7*w}
  })
}
...
```

Использует функцию для создания различных стилей из ширины (которая позднее изменится) и множителя

Присоединяет слушателя неподдерживаемого события к window

Удаляет слушателя неподдерживаемого события из window

Реализует специальную функцию для масштабирования переключателя на основании нового размера экрана

Вспомогательная функция `getStyle()` абстрагирует часть стилей, потому что в CSS встречаются повторения (`top`, `bottom`, `left` и `right`), но с разными значениями, зависящими от ширины окна. Соответственно, `getStyle()` получает значение и множитель `m` и возвращает значение в пикселах (числа в CSS-коде React интерпретируются как пикселы).

Остальной код понять несложно. Все, что нужно, — реализовать метод `render()`, который использует состояния и свойства для рендера четырех элементов `<div/>`. Каждый элемент имеет специальный стиль, определенный ранее в `constructor()`.

Листинг 6.12. Использование значений состояния для изменения размеров элементов

```
...
  render() {
    return <div>
      <div className="radio-tagger" style={this.state.taggerStyle}>
        <input type="radio" name={this.props.name} id={this.props.id}>
        </input>
        <label htmlFor={this.props.id}>
          <div className="radio-text" style={this.state.textStyle}>
          ➥ {this.props.label}</div>
          <div className="radio-outer" style={this.state.outerStyle}>
            <div className="radio-inner" style={this.state.innerStyle}>
              <div className="radio-selected"
              ➥ style={this.state.selectedStyle}>
              </div>
```

```
            </div>
          </div>
        </label>
      </div>
    </div>
  }
}
```

Вот и все, что касается реализации компонента React. Суть этого примера в том, что, используя события жизненного цикла в своих компонентах, вы можете создавать слушатели нестандартных событий. В рассмотренном примере это делалось при помощи `window`. Слушатели событий React работают аналогичным образом: React присоединяет события к `document`, как упоминалось в начале главы. И не забудьте удалить нестандартные слушатели событий в событии `unmount`!

Если вас интересуют масштабируемые переключатели и их реализации, не использующие React (то есть jQuery), я написал отдельное сообщение в блоге (http://mng.bz/kPMu) и создал демонстрационное приложение (http://jsfiddle.net/DSYz7/8). Конечно, вы найдете реализацию для React в архиве исходного кода этой книги.

А мы подходим к теме интеграции React с другими UI-библиотеками, такими как jQuery.

6.3. Интеграция React с другими библиотеками: UI-события jQuery

Как вы уже видели, React предоставляет в распоряжение разработчика стандартные события DOM; но что, если вам понадобится интегрироваться с другой библиотекой, использующей (инициирующей или прослушивающей) нестандартные события? Например, представьте, что у вас имеются компоненты jQuery, которые используют событие `slide` как элемент управления ползунок (slider). Требуется интегрировать виджет React в приложение jQuery. Для присоединения любых событий DOM, не предоставляемых React, можно воспользоваться событиями жизненного цикла `componentDidMount` и `componentWillUnmount`.

Вероятно, вы уже догадались по выбору событий жизненного цикла, что слушатель события будет присоединяться при подключении компонента и отсоединяться при его отключении. Отсоединение играет важную роль, чтобы потерянные слушатели событий не создавали конфликтов или проблем с быстродействием. (*Потерянными* называются обработчики событий, не имеющие узлов DOM, которые их создали, — потенциальная утечка памяти.)

Представьте, что вы работаете в компании, занимающейся потоковой передачей музыки, и вам поручено реализовать элемент управления громкостью в новой версии веб-проигрывателя (что-то вроде Spotify или iTunes). Нужно добавить надпись

и кнопки к ползунку, унаследованному из предыдущей версии (http://plugins.jquery.com/ui.slider).

Нужно реализовать надпись с числовым значением и две кнопки для уменьшения и увеличения значения на 1. При этом все фрагменты должны работать как единое целое: когда пользователь сдвигает ползунок влево или вправо, числовое значение и значения на кнопках должны измениться соответствующим образом. Если же пользователь щелкнет на любой из кнопок, ползунок тоже должен сместиться влево или вправо. Фактически нужно создать не просто ползунок, а виджет, изображенный на рис. 6.14.

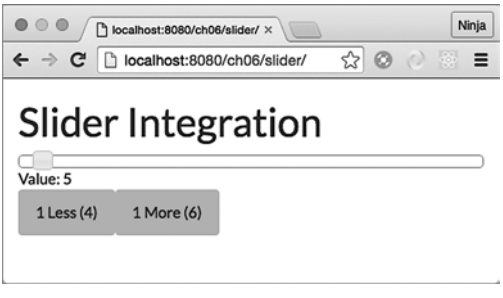


Рис. 6.14. Компоненты React (кнопки и текст «Value…») могут интегрироваться с другими библиотеками, например с элементом Slider из jQuery, чтобы разные элементы из разных библиотек взаимодействовали друг с другом

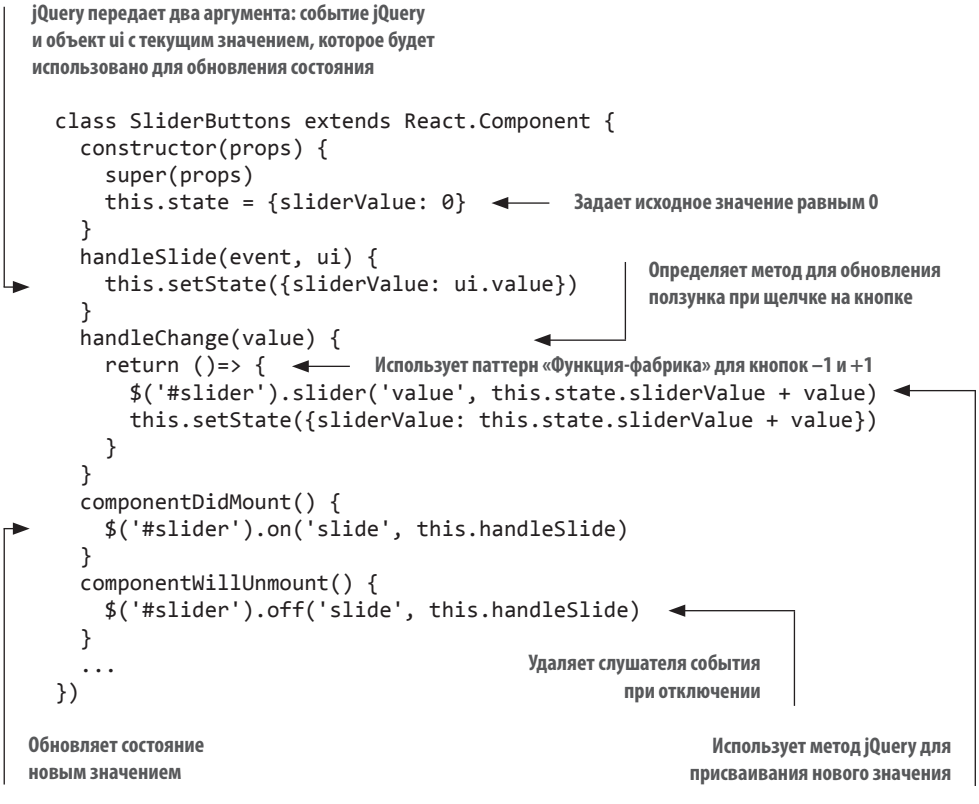
6.3.1. Интеграция кнопок

В области интеграции возможны два варианта: (1) присоединение событий для `Slider` из jQuery к компоненту React и (2) использование `window`. Начнем с первого варианта и используем его для кнопок.

> **ПРИМЕЧАНИЕ** Для этого способа интеграции кнопок характерны сильные связи. Объекты зависят друг от друга. В общем случае следует избегать решений с сильными связями. После того как мы рассмотрим это решение, будет реализовано другое решение для интеграции надписей с более слабыми связями.

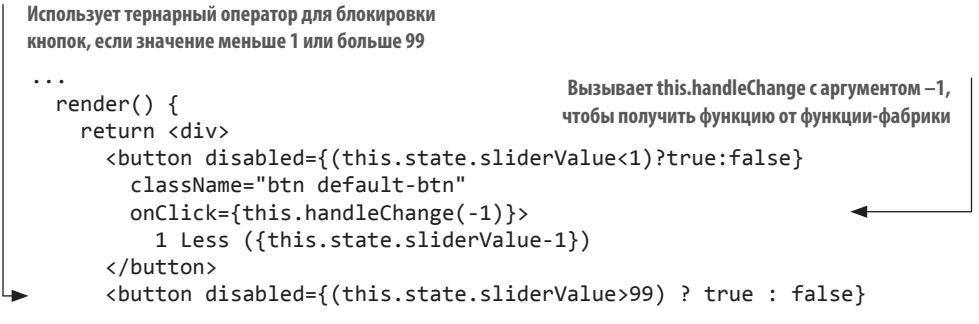
При возникновении события `slide` ползунка jQuery (указывающего на изменение значения) необходимо обновить значения кнопок (текст на кнопках). Можно присоединить слушателя события к ползунку jQuery в `componentDidMount` и организовать выполнение метода в компоненте React (`handleSlide`) при возникновении события `slide`. С каждым перемещением и изменением значения обновляется состояние (`sliderValue`). Этот подход реализован в компоненте `SliderButtons` из листинга 6.13 (ch06/slider/jsx/slider-buttons.jsx).

Листинг 6.13. Интеграция с плагином jQuery посредством событий



Метод `render()` класса `SliderButtons` содержит две кнопки с событиями `onClick`, динамический атрибут `disabled`, чтобы избежать присваивания значений меньших 0 (рис. 6.15) или выше 100, и классы Twitter Bootstrap для кнопок (ch06/slider/jsx/sliderbuttons.jsx).

Листинг 6.14. Рендер кнопок виджета



```
          className="btn default-btn"
          onClick={this.handleChange(1)}>
            1 More ({this.state.sliderValue+1})
        </button>
      </div>
    }
  })
```

Применяет классы Twitter Bootstrap, используя className

Рендерит следующее значение ползунка как надпись на кнопке

В результате, если значение выходит за границы заданного диапазона (минимум 0, максимум 100), кнопки блокируются. Например, когда значение равно 0, кнопка Less блокируется (рис. 6.15).

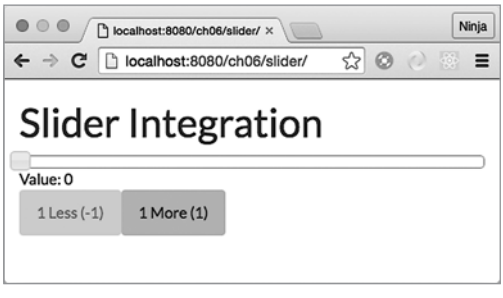


Рис. 6.15. Программная блокировка кнопки Less для предотвращения отрицательных значений

При перетаскивании ползунка текст на кнопках изменяется, а сами кнопки блокируются/разблокируются по мере необходимости. Благодаря обращению к ползунку в `handleChange()` щелчок на кнопках перемещает ползунок влево или вправо. А на следующем шаге мы займемся надписью Value, которая будет реализована компонентом React `SliderValue`.

6.3.2. Интеграция надписей

Вы знаете, как напрямую обращаться с вызовами к jQuery из методов React. В то же время jQuery можно отделить от React, используя другой объект для перехвата событий. Такая схема известна как паттерн «Слабая связь»; часто она является предпочтительной, потому что позволяет избежать лишних зависимостей. Иначе говоря, разным компонентам не обязательно знать подробности реализации друг друга. Компонент React `SliderValue` не знает, как обратиться с вызовом к ползунку jQuery, — и это хорошо, потому что позднее вам будет проще перейти с `Slider` на `Slider` версии 2.0 с другим интерфейсом.

Вы можете реализовать эту схему, передавая события `window` в событиях jQuery и определяя слушателей событий для `window` в методах жизненного цикла компонентов React. В листинге 6.15 приведен код `SliderValue` (ch06/slider/jsx/slider-value.jsx).

Листинг 6.15. Интеграция с плагином jQuery через window

```
class SliderValue extends React.Component {
  constructor(props) {
    super(props)
    this.handleSlide = this.handleSlide.bind(this)
    this.state = {sliderValue: 0}
  }
  handleSlide(event) {
    this.setState({sliderValue: event.detail.ui.value})
  }
  componentDidMount() {
    window.addEventListener('slide', this.handleSlide)
  }
  componentWillUnmount() {
    window.removeEventListener('slide', this.handleSlide)
  }
  render() {
    return <div className="" >
      Value: {this.state.sliderValue}
    </div>
  }
}
```

Присоединяет слушателя события slide к объекту window, чтобы инициировать выполнение handleSlide()

Отсоединяет slide от window, чтобы избежать появления потерянных обработчиков событий и утечек памяти

Кроме того, необходимо позаботиться о доставке нестандартного события. В первом решении (`SliderButtons`) это делать не требовалось, потому что вы использовали существующий плагин events. В этой реализации необходимо создать событие и передать его `window` с данными. Вы можете реализовать диспетчеры нестандартного события `slide` вместе с кодом, создающим объект jQuery `slider`, который представляет собой тег `script` в index.html (ch06/slider/index.html).

Листинг 6.16. Создание слушателей события в плагине jQuery

```
let handleChange = (e, ui)=>{
  var slideEvent = new CustomEvent('slide', {
    detail: {ui: ui, jQueryEvent: e} )
  })
  window.dispatchEvent(slideEvent)
}
$( '#slider' ).slider({
  'change': handleChange,
  'slide': handleChange
})
```

Передает данные jQuery с текущим значением ползунка

Создает обработчик события для ползунка jQuery, который будет передавать нестандартные события

Создает нестандартное событие

Передает событие window

Присоединяет слушателей событий для change (программное) и slide (UI)

Создает объект slider, используя контейнер с идентификатором slider

При выполнении этого кода и кнопки, и надпись будут работать безукоризненно. В решении были использованы два подхода: со слабыми и с сильными связями. Реализация второго короче, но первый вариант предпочтителен, потому что он упростит возможные изменения кода в будущем.

Как видно из этого примера интеграции, React хорошо работает в сочетании с другими библиотеками, прослушивая события в своем методе жизненного цикла `componentDidMount()`. React действует без предрассудков и прекрасно сотрудничает с другими! Простота интеграции React с другими библиотеками — огромное преимущество, потому что разработчики могут переключаться на React постепенно, вместо того чтобы переписывать все приложение с нуля, или же сколь угодно долго использовать свои любимые старые библиотеки с React.

6.4. Вопросы

1. Выберите правильный синтаксис объявления события: `onClick=this.doStuff`, `onclick={this.doStuff}`, `onClick="this.doStuff"`, `onClick={this.doStuff}` или `onClick={this.doStuff()}`.
2. Событие `componentDidMount()` не инициируется во время серверного рендеринга компонента React, для которого оно было объявлено. Да или нет?
3. Один из способов обмена информацией между дочерними компонентами заключается в передаче объекта родителю. Да или нет?
4. По умолчанию `event.target` может использоваться асинхронно и вне обработчика события. Да или нет?
5. Возможна интеграция со сторонними библиотеками и событиями, не поддерживаемыми React. Для этого следует создать слушатели для событий жизненного цикла компонентов. Да или нет?

6.5. Итоги

- Обработчик `onClick` предназначен для получения щелчков кнопки мыши и сенсорной панели (трекпада).
- Для слушателей событий используется синтаксис JSX `<a onNAME={this.METHOD}>`.
- Выполните связывание обработчиков событий вызовом `bind()` в `constructor()` или в JSX, если вы хотите использовать `this` в обработчике события как значение экземпляра класса компонента.
- Событие `componentDidMount()` инициируется только в браузере. Событие `componentWillMount()` инициируется как в браузере, так и на стороне сервера.

- React поддерживает большинство стандартных событий HTML DOM, предоставляя и используя синтетические объекты событий.
- `componentDidMount()` и `componentWillUnmount()` могут использоваться для интеграции React с другими фреймворками и событиями, не поддерживаемыми React.

6.6. Ответы

1. Синтаксис `onClick={this.doStuff}` верен, потому что `onClick` должно передаваться только определение функции, а не ее вызов (результат вызова, если выражаться точно).
2. Да. `componentDidMount()` вышлигается только для React в браузере, но не на стороне сервера. По этой причине разработчики используют `componentDidMount()` для запросов AJAX/XHR. За информацией о событиях жизненного цикла компонентов обращайтесь к главе 5.
3. Да. Перемещение данных вверх по иерархии компонентов позволяет передавать их разным дочерним компонентам.
4. Нет. Этот объект используется повторно, поэтому вы не сможете использовать его в асинхронной операции без вызова `persist()` для `SyntheticEvent`.
5. Да. События жизненного цикла компонентов — одно из самых подходящих мест для этой цели, потому что вы можете выполнить подготовку до того, как компонент активизируется, и непосредственно перед его удалением.

7 Работа с формами в React

Посмотрите вступительный видеоролик к этой главе, отсканировав QR-код или перейдя по ссылке http://reactquickly.co/videos/ch07.

ЭТА ГЛАВА ОХВАТЫВАЕТ СЛЕДУЮЩИЕ ТЕМЫ:

- Рекомендуемый способ работы с формами в React.
- Отслеживание изменений в формах.
- Использование ссылок на данные доступа.
- Альтернативные способы работы с формами.
- Установка значений по умолчанию для элементов формы.

К настоящему моменту вы узнали о событиях, состояниях, компонентах и других важных концепциях и возможностях React. Но помимо перехвата пользовательских событий я еще не рассказал, как получить текст и ввод от других элементов форм: текстовых полей, текстовых областей и переключателей. Работа с этими элементами чрезвычайно важна для веб-разработки, потому что эти элементы позволяют приложениям получать данные (например, текст) и действия (например, щелчки) от пользователей.

В этой главе задействованы практически все темы, упоминавшиеся до настоящего момента. Вы постепенно начнете видеть, как все фрагменты складываются в единое целое.

ПРИМЕЧАНИЕ Исходный код примеров этой главы доступен по адресам www.manning.com/books/react-quickly и https://github.com/azat-co/react-quickly/tree/master/ch07 (в папке ch07 репозитория GitHub https://github.com/azat-co/react-quickly). Также некоторые демонстрационные примеры доступны по адресу http://reactquickly.co/demos.

7.1. Рекомендуемый способ работы с формами в React

В обычной разметке HTML при работе с элементами ввода модель DOM страницы хранит текущее значение этого элемента в узле DOM. К значению можно обратиться такими методами, как `document.getElementById('email').value`, или при помощи методов jQuery. Фактически модель DOM становится хранилищем данных.

В React при работе с формами или любыми другими полями ввода (например, автономными текстовыми полями или кнопками) возникает интересная проблема. В документации React сказано: «Компоненты React должны представлять состояние представления в любой момент времени, не только на момент инициализации». Вся суть React — простота за счет применения декларативного стиля для описания пользовательских интерфейсов. React описывает пользовательский интерфейс — его конечную стадию, то, как он должен выглядеть.

Заметили противоречие? В традиционных элементах форм HTML состояния элементов изменяются с вводом данных пользователем. С другой стороны, React использует декларативный подход для описания пользовательских интерфейсов. Чтобы правильно отражать состояние элемента, ввод должен быть динамическим.

Таким образом, при *отказе* от поддержания состояния компонента (в JavaScript) и его синхронизации с представлением появляются проблемы — возможны ситуации, в которых внутреннее состояние отличается от представления. React не знает об изменении состояния. Это может привести к всевозможным проблемам и ошибкам, и противоречит простой философии React. Лучше всего поддерживать метод `render()` в React как можно ближе к реальной модели DOM — и это относится к данным в элементах форм.

Рассмотрим пример поля для ввода текста. Новое значение должно быть включено в `render()` для этого компонента. Соответственно, вы должны присвоить новое значение элементу с использованием `value`. Но если вы реализуете поле `<input>` в HTML, React всегда будет синхронизировать `render()` с реальной моделью DOM. React не позволит пользователю изменить значение. Попробуйте — и убедитесь сами. Выглядит странно, но это правильное поведение для React!

```
render() {
  return <input type="text" name="title" value="Mr." />
}
```

Этот код отражает представление в любом состоянии, поэтому его значением всегда будет `Mr.`. С другой стороны, поля ввода должны изменяться в зависимости от щелчков или ввода текста пользователем. С учетом всего сказанного сделаем значение динамическим. Такая реализация будет более совершенной, потому что она будет обновляться в зависимости от состояния:

```
render() {
  return <input type="text" name="title" value={this.state.title} />
}
```

А как же значение `state`? React ничего не знает о том, что пользователь вводит текст в элементах формы. Необходимо реализовать обработчик события для перехвата изменений в `onChange`:

```
handleChange(event) {
  this.setState({title: event.target.value})
}
render() {
  return <input type="text" name="title" value={this.state.title}
  ➥ onChange={this.handleChange.bind(this)}/>
}
```

С учетом этих обстоятельств лучше всего принять следующие меры для синхронизации внутреннего состояния с представлением (рис. 7.1):

1. Определить элементы в `render()` с использованием значений из `state`.
2. Сохранить изменения в элементе формы по мере их возникновения с использованием `onChange`.
3. Обновить внутреннее состояние в обработчике состояния.
4. Новые значения сохраняются в `state`, после чего представление обновляется новым вызовом `render()`.

Рис. 7.1. Правильный способ работы с элементами формы: от пользовательского ввода к событиям, затем к состоянию и представлению

На первый взгляд может показаться, что это требует довольно значительных усилий, но я надеюсь, что по мере накопления опыта использования React вы оцените такой подход. Он называется *односторонним связыванием*, потому что состояние изменяет представления — и всё. В обратную сторону это не работает: только одностороннее движение от состояния к представлению. При одностороннем связывании библиотека не будет автоматически обновлять состояние (или модель). Одно из главных преимуществ одностороннего связывания заключается в том, что оно

снижает сложность при работе с большими приложениями, в которых множественные представления могут неявно обновлять множественные состояния (модели данных) и наоборот (рис. 7.2).

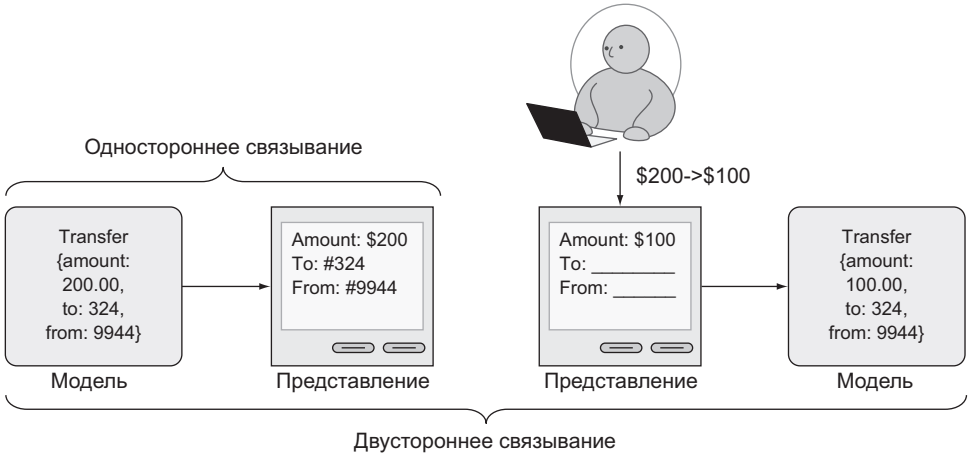


Рис. 7.2. Одностороннее связывание отвечает за переход «модель->представление». Двустороннее связывание также обрабатывает переход изменений от представления к модели

Простота не всегда означает, что вам приходится писать меньше кода. Иногда в таких ситуациях приходится писать лишний код для ручного присваивания данных из обработчиков событий в состояние (которое визуализируется в представлении), но такой подход становится более эффективным в том случае, когда это касается сложных пользовательских интерфейсов и одностраничных приложений с мириадами представлений и состояний. «Просто» не всегда означает «легко».

И наоборот, двустороннее связывание позволяет представлениям автоматически изменять состояния без явной реализации процесса. Двустороннее связывание лежит в основе работы Angular 1. Интересно, что в Angular 2 была позаимствована и используется по умолчанию концепция одностороннего связывания из React (хотя вы можете явно выбрать двустороннее связывание).

По этой причине я сначала рассмотрю рекомендованный способ работы с формами. Он называется *использованием управляемых компонентов* и гарантирует, что внутреннее состояние компонентов всегда синхронизировано с представлением. Управляемые элементы форм называются так, потому что React управляет или задает значения. Альтернативное решение — неуправляемые компоненты, которые будут рассмотрены в разделе 7.2.

Вам уже известен рекомендованный способ работы с полями ввода в React: отслеживание изменений и применение их к состоянию, как показано на рис. 7.1

(от ввода к измененному представлению). А теперь посмотрим, как определить форму и ее элементы.

7.1.1. Определение формы и ее событий в React

Начнем с элемента `<form>`. Как правило, элементы ввода не должны хаотично располагаться в произвольных местах DOM. При наличии большого числа функционально разных наборов элементов ввода ситуация может выйти из-под контроля. Вместо этого элементы ввода, предназначенные для одной цели, упаковываются в элемент `<form></form>`.

Обертка `<form>` не является обязательной. В простых пользовательских интерфейсах ничто не мешает использовать элементы форм сами по себе. В более сложных интерфейсах, где на одной странице могут размещаться несколько групп элементов, стоит использовать `<form>` для каждой такой группы. В React `<form>` рендерится так же, как `<form>` из разметки HTML, поэтому правила, которыми вы руководствуетесь при работе с формами HTML, также будут применимы и к элементу `<form>` в React. Например, согласно спецификации HTML5 *не допускается* использование вложенных форм[1].

Элемент `<form>` может обладать событиями. React поддерживает три события для форм в дополнение к стандартным событиям React DOM (см. табл. 6.1):

- `onChange` — инициируется при изменении любых элементов ввода на форме.
- `onInput` — инициируется при каждом изменении значений элементов `<textarea><input>`. Команда React не рекомендует использовать этот метод (см. врезку).
- `onSubmit` — инициируется при отправке формы, обычно нажатием Enter.

ONCHANGE И ONINPUT

Событие React onChange инициируется при каждом изменении — в отличие от события change DOM (http://mng.bz/lJ37), которое может не срабатывать при каждом изменении значения, но срабатывает при потере фокуса. Например, для `<input type="text">` пользователь может набирать данные без onChange; только после того, как пользователь нажмет Tab или перейдет к другому элементу щелчком мыши, в HTML будет инициировано событие `onChange` (обычное событие браузера). Как упоминалось ранее, в React `onChange` инициируется при каждом нажатии клавиши, не только при потере фокуса. С другой стороны, onInput в React является оберткой для события DOM onInput, которое инициируется при каждом изменении.

[1] В спецификации сказано, что контент должен быть потоковым, но не должен иметь дочерних элементов <form>. См. www.w3.org/TR/html5/forms.html#the-form-element.

В результате событие React `onChange` работает не так, как событие `onChange` в HTML: оно более логично и имеет больше общего с событием HTML onInput. Разработчикам рекомендуется использовать onChange в React и выбирать `onInput` только тогда, когда вам нужно обратиться к встроенному поведению для события onInput. Дело в том, что поведение обертки React `onChange` более логично и последовательно.

В дополнение к трем уже упомянутым событиям, `<form>` может обрабатывать такие стандартные события React, как `onKeyUp` и `onClick`. События форм могут пригодиться для перехвата конкретного события всей формы (то есть группы элементов ввода).

Например, для повышения качества пользовательского интерфейса будет полезно дать пользователю возможность отправлять данные нажатием клавиши Enter (предполагается, что текстовая область (textarea) не является текущим элементом — в этом случае нажатие Enter должно создавать новую строку). Вы можете прослушивать событие отправки формы, создав слушателя события, который инициирует вызов `this.handleSubmit()`:

```
handleSubmit(event) {
  ...
}
render() {
  <form onSubmit={this.handleSubmit}>
    <input type="text" name="email" />
  </form>
}
```

ПРИМЕЧАНИЕ Функция `handleSubmit()` должна быть реализована за пределами `render()`, как это было бы сделано для любого другого события. React не требует особых правил назначения имен, так что вы можете присвоить обработчику событий любое имя на свое усмотрение (при условии, что имя будет понятным и в какой-то степени последовательным). В этой книге я придерживаюсь самой популярной схемы: обработчики событий снабжаются префиксом `handle`, чтобы они отличались от обычных методов классов.

ПРИМЕЧАНИЕ На всякий случай напомню: не вызывайте метод (не ставьте круглые скобки) и не заключайте фигурные скобки в двойные кавычки (правильно: `СОБЫТИЕ={this.МЕТОД}`) при назначении обработчика события. Для некоторых читателей это одна из простейших конструкций JavaScript, но вы не поверите, сколько раз я видел ошибки, связанные с непониманием двух обстоятельств в коде React: передается определение функции, а не ее результат, и фигурные скобки используются как значения атрибутов JSX.

Другой способ реализации отправки данных форм по нажатии Enter основан на ручном прослушивании события отпускания клавиши (`onKeyUp`) и проверке кода клавиши (13 для Enter):

```
handleKeyUp(event) {
  if (event.keyCode == 13) return this.sendData()
}
render() {
  return <form onKeyUp={this.handleKeyUp}>
  ...
  </form>
}
```

Следует обратить внимание на то, что метод `sendData()` реализуется где-то в другой точке класса/компонента. Кроме того, для работы `this.sendData()` необходимо использовать `bind(this)` для связывания контекста с обработчиком события в `constructor()`.

Подведем итог: события можно назначать на уровне элемента формы, а не только для отдельных элементов форм. Теперь посмотрим, как определяются элементы форм.

7.1.2. Определение элементов форм

Почти все поля ввода в HTML реализуются всего четырьмя элементами: `<input>`, `<textarea>`, `<select>` и `<option>`. Вы еще не забыли, что в React свойства неизменяемы? Что же, элементы форм особенные, потому что пользователи должны взаимодействовать с элементами и изменять эти свойства. Для всех остальных элементов это невозможно.

Чтобы придать этим элементам особый статус, в React они наделяются изменяемыми свойствами `value`, `checked` и `selected`. Эти специальные изменяемые свойства также называются *интерактивными свойствами*.

> **ПРИМЕЧАНИЕ** React DOM также поддерживает другие элементы, относящиеся к построению форм, такие как `<keygen>`, `<datalist>`, `<fieldset>` и `<label>`. Эти элементы не обладают такими суперспособностями, как изменяемый атрибут/свойство value, они рендерятся как соответствующие теги HTML. По этой причине в книге мы ограничимся только четырьмя главными элементами, наделенными суперспособностями.

Ниже перечислены интерактивные свойства/поля (те, что могут изменяться), которые могут читаться из таких событий, как `onChange`, присоединенных к элементам форм (см. раздел 6.1.3):

- `value` — применяется к элементам `<input>`, `<textarea>` и `<select>`.
- `checked` — применяется к элементам `<input>` с `type="checkbox"` и `type="radio"`.
- `selected` — применяется к `<option>` (используется с `<select>`).

Вы можете читать их значения и изменять их, работая с этими интерактивными (изменяемыми) свойствами. Рассмотрим несколько примеров того, как определяется каждый из этих элементов.

Элемент <INPUT>

Элемент `<input>` рендерит различные поля, используя разные значения для атрибута `type`:

- `text` — простое поле для ввода текста.
- `password` — поле для ввода замаскированного текста (для сохранения конфиденциальности).
- `radio` — кнопка-переключатель. Используйте одинаковые имена для создания группы переключателей.
- `checkbox` — флажок. Используйте одинаковые имена для создания группы.
- `button` — элемент формы «кнопка».

Все эти элементы типа `<input>`, кроме флажков и переключателей, используются прежде всего для работы с `value` как интерактивным/изменяемым свойством элемента. Например, поле для ввода адреса электронной почты может использовать состояние `email` и обработчик события `onChange`:

```
<input
  type="text"
  name="email"
  value={this.state.email}
  onChange={this.handleEmailChange}/>
```

Среди полей ввода есть два исключения, у которых `value` не является основным изменяемым атрибутом, — это типы `checkbox` и `radio`. Они используют значение `checked`, потому что эти два типа имеют одно значение на элемент HTML, а следовательно, значение не изменяется — но изменяется состояние `checked`/`selected`. Например, вы можете определить три переключателя в одной группе (`radioGroup`), как показано на рис. 7.3.

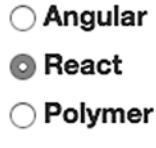


Рис. 7.3. Группа переключателей

Как упоминалось ранее, значения (`value`) жестко фиксируются, потому что изменять их не нужно. С действиями изменяется атрибут `checked` элемента, как показано в листинге 7.1 (ch07/elements/jsx/content.jsx).

Листинг 7.1. Рендеринг переключателей и обработка изменений

```
class Content extends React.Component {
  constructor(props) {
    super(props)
    this.handleRadio = this.handleRadio.bind(this)
    ...
    this.state = {
      ...
      radioGroup: {
        angular: false,
```

```
        react: true,
        polymer: false
      }
    }
  }
  handleRadio(event) {
    let obj = {} // Стереть другие переключатели
    obj[event.target.value] = event.target.checked // true
    this.setState({radioGroup: obj})
  }
  ...
  render() {
    return <form>
      <input type="radio"
        name="radioGroup"
        value='angular'
        checked={this.state.radioGroup['angular']}
        onChange={this.handleRadio}/>
      <input type="radio"
        name="radioGroup"
        value='react'
        checked={this.state.radioGroup['react']}
        onChange={this.handleRadio}/>
      <input type="radio"
        name="radioGroup"
        value='polymer'
        checked={this.state.radioGroup['polymer']}
        onChange={this.handleRadio}/>
      ...
    </form>
  }
}
```

Назначает установленный переключатель по умолчанию в состоянии

Использует атрибут target.checked для получения логического значения, показывающего, выбран ли данный переключатель

Использует атрибут из объекта state или любого атрибута state

Использует тот же обработчик события onChange, потому что значение переключателя можно получить из target.value

Для флажков применяется аналогичный подход: использование атрибута `checked` и логические значения для состояний. Эти логические значения могут храниться в состоянии `checkboxGroup`:

```
class Content extends React.Component {
  constructor(props) {
    super(props)
    this.handleCheckbox = this.handleCheckbox.bind(this)
    // ...
    this.state = {
      // ...
      checkboxGroup: {
      node: false,
      react: true,
      express: false,
      mongodb: false
    }
  }
}
```

Затем обработчик события (который связывается в конструкторе) получает текущие значения, добавляет `true` или `false` из `event.target.value` и задает состояние:

```
handleCheckbox(event) {
  let obj = Object.assign(this.state.checkboxGroup)
  obj[event.target.value] = event.target.checked   <-- true или false
  this.setState({checkboxGroup: obj})
}
```

В присваивании из состояния в `radio` нет необходимости, потому что переключатели могут иметь только одно выбранное значение. А это означает, что вы можете использовать пустой объект. С флажками дело обстоит иначе: в группе может быть выбрано несколько значений, поэтому нужна операция слияния, а не замены.

В JavaScript объекты передаются и присваиваются по ссылке. Таким образом, в команде `obj = this.state.checkboxGroup obj` в действительности является состоянием. Напомню, что состояние не должно изменяться напрямую. Для предотвращения любых потенциальных конфликтов лучше присваивать значение вызовом `Object.assign()`. Этот прием также называется *клонированием*. Другой, менее эффективный и менее надежный способ присваивания основан на присваивании по значению с использованием JSON:

```
clonedData = JSON.parse(JSON.stringify(originalData))
```

Если вы используете массивы состояний вместо объектов и вам потребовалось выполнить присваивание по значению, используйте конструкцию `clonedArray = Array.from(originArray)` или `clonedArray = originArray.slice()`.

Обработчик события `handleCheckbox()` может использоваться для получения значения из `event.target.value`. В листинге 7.2 приведена разметка `render()` (ch07/elements/jsx/content.jsx), которая использует значения состояния для четырех флажков (рис. 7.4).

Рис. 7.4. Рендеринг флажков с выбором варианта React по умолчанию

Листинг 7.2 Определение флажков

```
<input type="checkbox"
  name="checkboxGroup"
  value='node'
  checked={this.state.checkboxGroup['node']}
  onChange={this.handleCheckbox}/>
<input type="checkbox"
  name="checkboxGroup"
  value='react'
  checked={this.state.checkboxGroup['react']}
  onChange={this.handleCheckbox}/>
<input type="checkbox"
```

Использует состояние как значение. Это может быть атрибут объекта или просто атрибут state

Использует onChange для отслеживания действий пользователя

```
  name="checkboxGroup"
  value='express'
  checked={this.state.checkboxGroup.express}
  onChange={this.handleCheckbox}/>
<input type="checkbox"
  name="checkboxGroup"
  value='mongodb'
  checked={this.state.checkboxGroup['mongodb']}
  onChange={this.handleCheckbox}/>
```

Использует «точечную запись», когда ключи являются действительными именами JS

Выполнять связывание в элементе не нужно, потому что оно выполняется в конструкторе (истинно для всех флажков)

Фактически при использовании флажков или переключателей можно жестко присвоить значение в каждом отдельном элементе и использовать `checked` как изменяемый атрибут. А теперь посмотрим, как работать с другими элементами ввода.

Элемент <TEXTAREA>

Элементы `<textarea>` предназначены для получения и вывода длинных текстовых сообщений — заметок, сообщений в блогах, фрагментов кода и т. д. В обычном HTML `<textarea>` использует внутреннюю разметку HTML (то есть дочерние элементы) в качестве значения:

```
<textarea>
  With the right pattern, applications...
</textarea>
```

Пример показан на рис. 7.5.

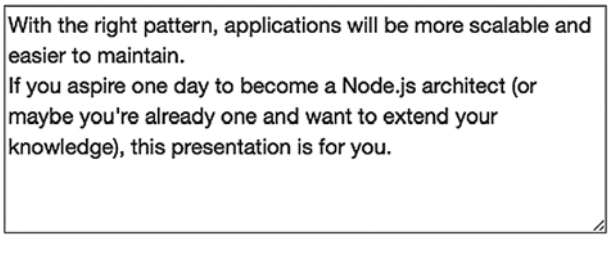


Рис. 7.5. Определение и рендер элемента <textarea>

С другой стороны, React использует *атрибут* `value`. С учетом этого обстоятельства использование внутренней разметки HTML/текста в качестве значения становится антипаттерном. React преобразует все дочерние элементы `<textarea>` (если они используются) в значение по умолчанию (подробнее о значениях по умолчанию в разделе 7.2.4):

```
<!-- Антипаттерн: НЕ ДЕЛАЙТЕ так! -->
<textarea name="description">{this.state.description}</textarea>
```

Вместо этого для `<textarea>` рекомендуется использовать атрибут (или свойство) `value`:

```
render() {
  return <textarea name="description" value={this.state.description}/>
}
```

Чтобы прослушивать изменения, используйте `onChange` как для элементов `<input>`.

Элементы <SELECT> и <OPTION>

Поля `<select>` и `<option>` очень удобны для выбора одного или нескольких значений из заранее заполненного списка значений. Список значений скрывается за элементом до того момента, как пользователь раскроет его (для одиночного выбора), как показано на рис. 7.6.

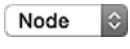


Рис. 7.6. Рендеринг и предварительный выбор значения в раскрывающемся списке

`<select>` — еще один элемент, поведение которого в React отличается от обычной разметки HTML. Например, в обычной разметке HTML для получения индекса выбранного элемента можно использовать `selectDOMNode.selectedIndex` или `selectDOMNode.selectedOptions`. В React для `<select>` используется `value` как в листинге 7.3 (ch07/elements/jsx/content.jsx).

Листинг 7.3. Рендер элементов формы

```
...
constructor(props) {
  super(props)
  this.state = {selectedValue: 'node'}
}
handleSelectChange(event) {
  this.setState({selectedValue: event.target.value})
}
...
render() {
  return <form>
    <select
      value={this.state.selectedValue}
      onChange={this.handleSelectChange}>
        <option value="ruby">Ruby</option>
        <option value="node">Node</option>
        <option value="python">Python</option>
    </select>
  </form>
}
...
```

Этот фрагмент рендерит раскрывающееся меню и предварительно выбирает значение `node` (которое должно быть задано в конструкторе, как показано на рис. 7.6).

Иногда бывает нужно использовать элемент с множественным выделением. Это можно сделать в JSX/React, передавая атрибут `multiple` без значения (React по умолчанию использует `true`) или со значением `{true}`.

> **СОВЕТ** Помните, что для единства стиля и для предотвращения недоразумений я рекомендую заключать все логические значения в фигурные скобки `{}`, а не `""`. Конечно, `"true"` и `{true}` выдают одинаковые результаты. Тем не менее `"false"` также выдаст `true`. Дело в том, что строка `"false"` в JavaScript интерпретируется как истинное значение (истинность).

Чтобы осуществить предварительное выделение в списке нескольких элементов, можно передать `<select>` массив вариантов через атрибут `value`. Например, следующий код предварительно выделяет `Meteor` и `React`:

```
<select multiple={true} value={['meteor', 'react']}>
  <option value="meteor">Meteor</option>
  <option value="react">React</option>
  <option value="jQuery">jQuery</option>
</select>
```

С `multiple={true}` генерируется элемент с множественным выделением, в котором предварительно выделены значения `Meteor` и `React` (рис. 7.7).

Рис. 7.7. Рендеринг и предварительное выделение в списке с множественным выделением

В целом определение элементов форм в React мало чем отличается от определения их в обычной разметке HTML, разве что значение `value` используется чаще. Мне нравится эта логическая целостность. Однако определение — всего лишь половина дела; другая половина — получение значений. Мы уже понемногу занимались этим в предыдущих примерах. А теперь займемся перехватом событий.

7.1.3. Отслеживание изменений в формах

Как упоминалось ранее, чтобы отслеживать изменения в элементах форм, вы определяете слушателя события `onChange`. Это событие перекрывает нормальное событие DOM `onInput`. Другими словами, если вам понадобится обычное

поведение `onInput` из HTML DOM, вы можете использовать событие `onInput` из React. С другой стороны, событие React `onChange` не полностью эквивалентно обычному событию DOM `onChange`. Обычное событие DOM `onChange` может инициироваться только при потере фокуса элементом, тогда как событие React `onChange` инициируется при любом новом вводе. Условие срабатывания `onChange` зависит от конкретного элемента:

- `<input>`, `<textarea>` и `<select>` — `onChange` инициируется изменением `value`.
- `<input>` с типом `checkbox` или `radio` — `onChange` инициируется изменением `checked`.

Соответственно, изменяется и способ чтения значения. В аргументе обработчика события вы получаете объект `SyntheticEvent`. У него есть свойство `target` со значением `value`, `checked` или `selected` в зависимости от элемента.

Чтобы прослушивать изменения, вы определяете обработчик события где-то в своем компоненте (его также можно определить во встроенном формате, то есть в фигурных скобках JSX `{}`) и создать атрибут `onChange`, указывающий на обработчик события. Например. следующий код отслеживает изменения в поле для ввода адресов электронной почты (ch07/elements/jsx/content.jsx).

Листинг 7.4. Генерирование элементов форм и отслеживание изменений

```
handleChange(event) {
  console.log(event.target.value)
}
render() {
  return <input
    type="text"
    onChange={this.handleChange}
    defaultValue="hi@azat.co"/>
}
```

Интересно, что если вы не определите `onChange`, но предоставите `value`, React выдаст предупреждение и сделает элемент доступным только для чтения. Если вы намерены создать поле только для чтения, лучше определить его явно при помощи `readOnly`. Тем самым вы не только устраните предупреждение, но и гарантируете, что другие программисты, читающие этот код, будут знать, что поле намеренно было сделано доступным только для чтения. Чтобы явно задать значение, присвойте `readOnly` `true` (то есть `readOnly={true}`) или добавьте атрибут `readOnly` без значения; React по умолчанию добавит значение `true` к атрибуту.

Обнаруженные изменения в элементах можно сохранить в состоянии компонента:

```
handleChange(event) {
  this.setState({emailValue: event.target.value})
}
```

Рано или поздно эту информацию потребуется передать серверу или другому компоненту. В этом случае значения будут аккуратно упорядочены в состоянии.

Представьте, что вы создаете форму заявки на кредит с именем, адресом, номером телефона и номером социального страхования пользователя. Каждое поле обрабатывает свои изменения. В нижней части формы находится кнопка Submit для отправки состояния серверу. В листинге 7.5 показано поле имени с onChange, которое сохраняет весь ввод в состоянии (ch07/elements/jsx/content.jsx).

Листинг 7.5. Рендеринг элементов формы

```
constructor(props) {
  super(props)
  this.handleInput = this.handleInput.bind(this)
  this.handleSubmit = this.handleSubmit.bind(this)
  ...
}
handleFirstNameChange(event) {
  this.setState({firstName: event.target.value})
}
...
handleSubmit() {
    fetch(this.props['data-url'], {method: 'POST', body:
     ➥ JSON.stringify(this.state)})
      .then((response)=>{return response.json()})
      .then((data)=>{console.log('Submitted: ', data)})
}
render() {
  return <form>
    <input name="firstName"
      onChange={this.handleFirstNameChange}
      type="text"/>
        ...
      <input
        type="button"
        onClick={this.handleSubmit}
        value="Submit"/>
  </form>
}
```

Сохраняет изменения в поле firstName в состоянии

Отправляет данные по URL-адресу из свойства data-url с использованием браузерного Fetch API на базе обещаний (на момент написания книги считается экспериментальным, но поддерживается большинством современных браузеров)

Определяет обработчик события для кнопки Submit

ПРИМЕЧАНИЕ Fetch — экспериментальный браузерный метод для выполнения запросов AJAX/XHR на базе обещаний (promises). О его использовании и поддержке (он поддерживается большинством современных браузеров на момент написания книги) можно узнать по адресу http://mng.bz/mbMe.

Итак, вы научились определять элементы, отслеживать изменения при помощи событий и обновлять состояние (которое используется для вывода значений). В следующем разделе будет рассмотрен пример.

7.1.4. Поле для банковского счета

Продолжим пример с формой заявки на кредит. После того как заявка будет одобрена, пользователь должен иметь возможность ввести номер банковского счета, на который должны быть переведены деньги. Реализуем компонент поля для банковского счета, используя новые знания. Это будет управляемый элемент, считающийся предпочтительным при работе с формами в React.

В компоненте, показанном в листинге 7.6 (ch07/account/jsx/content.jsx), представлено поле для ввода номера счета. Оно должно принимать только числовые данные (рис. 7.8). Чтобы ограничить ввод цифрами 0–9, можно воспользоваться управляемым компонентом для подавления всех значений, кроме цифр. Обработчик события задает состояние только после фильтрации ввода.

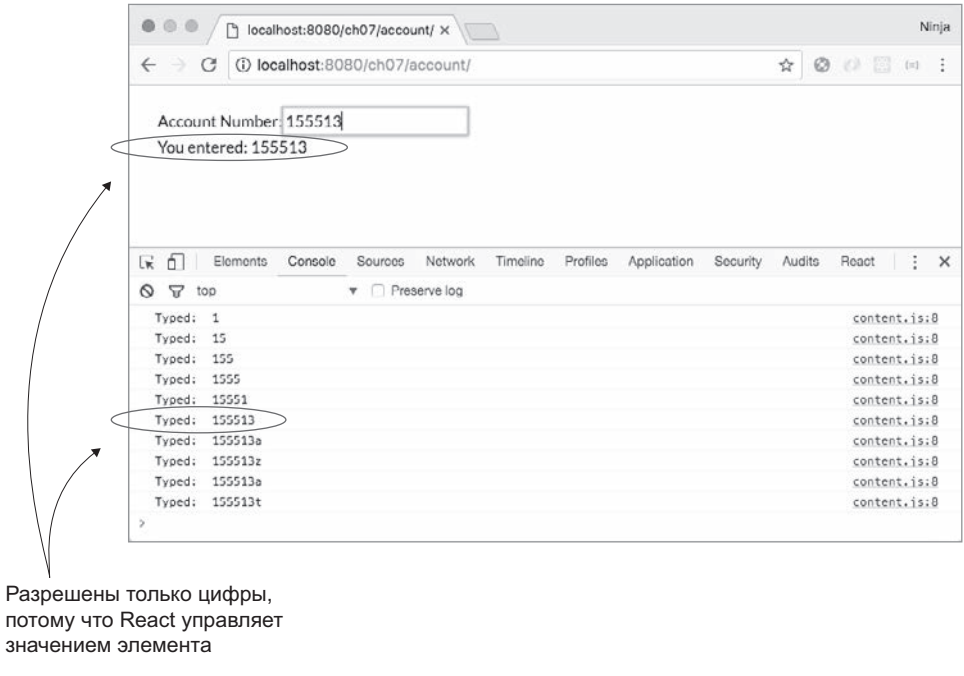


Рис. 7.8. Как видно из консольного вывода, в поле можно вводить любые данные. Тем не менее в качестве значения и в представлении допустимы только цифры, потому что элемент является управляемым

Листинг 7.6. Реализация управляемого компонента

```
class Content extends React.Component {
  constructor(props) {
    super(props)
    this.handleChange = this.handleChange.bind(this)
    this.state = {accountNumber: ''}
```

Задает пустую строку как исходный номер банковского счета

```
  }
  handleChange(event) {
    console.log('Typed: ', event.target.value)
    this.setState({accountNumber: event.target.value.replace(/[^0-9]/ig,
    ➥ '')})
  }
  render() {
    return <div>
      Account Number:
      <input
        type="text"
        onChange={this.handleChange}
        placeholder="123456"
        value={this.state.accountNumber}/>
      <br/>
      <span>{this.state.accountNumber.length>0?'You entered: ' +
       ➥ this.state.accountNumber: ''}</span>
    </div>
  }
})
```

Выводит нефильтрованное значение в том виде, в каком оно было введено

Фильтрует значение и обновляет состояние

Отслеживает изменения

Управляет элементом, присваивая значение state

Выводит номер счета, если он не пуст. «length» — строковое свойство, возвращающее количество символов. Если значение пустое, ничего не выводится

Для удаления всех символов, не являющихся цифрами, используется регулярное выражение (http://mng.bz/r7sq) `/[^0-9]/ig` и строковая функция `replace` (http://mng.bz/2Qon). Конструкция `replace(/[^0-9]/ig, '')` — несложный вызов функции регулярных выражений, который заменяет все, кроме цифр, пустой строкой. Модификатор `ig` означает игнорирование регистра символов и глобальность (иначе говоря, ищутся все совпадения).

В `render()` присутствует поле ввода, которое является управляемым компонентом из-за `value={this.state.accountNumber}`. Если вы запустите этот пример, в поле будут вводиться только числовые данные, потому что React присваивает новому состоянию результат фильтрации введенных данных (рис. 7.9).

Следуя общепринятым правилам React по работе с элементами ввода и формами, вы можете реализовать проверку данных и добиться того, чтобы представление соответствовало требованиям.

> **ПРИМЕЧАНИЕ** Очевидно, в компоненте для ввода банковского счета реализуется проверка клиентской части, которая не помешает хакеру внедрить вредоносные данные в запрос XHR, отправленный серверу. А следовательно, вы должны обеспечить необходимую проверку на уровне серверной части и/или бизнес-уровне — например, ORM/ODM (https://ru.wikipedia.org/wiki/ORM).

Итак, теперь вам известен основной подход к работе с формами: создание управляемых компонентов. А теперь рассмотрим некоторые альтернативы.

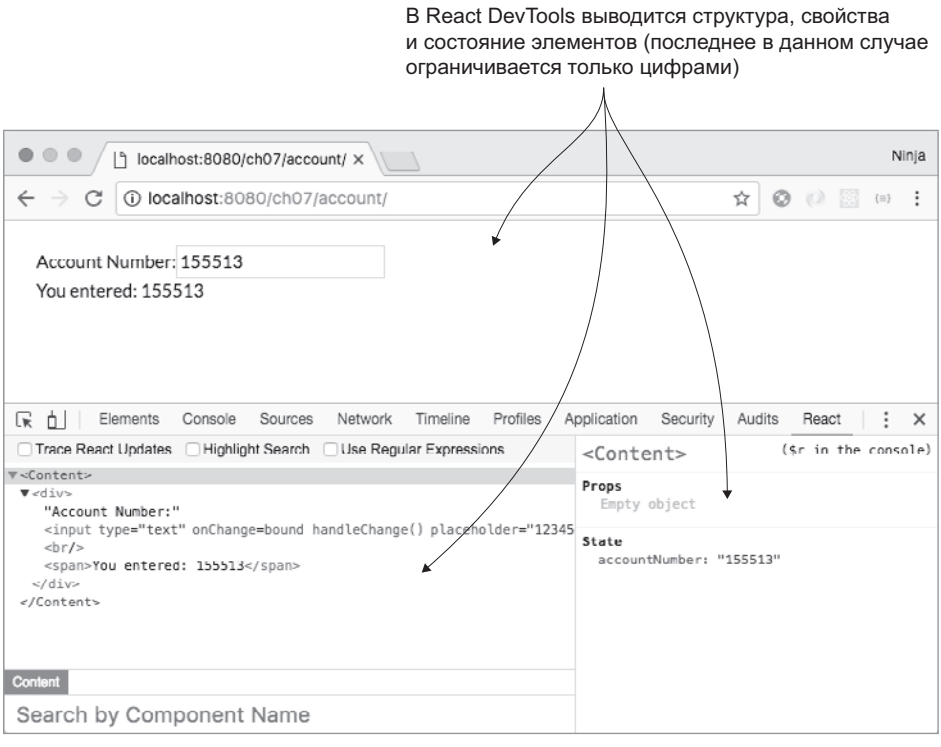


Рис. 7.9. Управляющие элементы фильтруют ввод, сохраняя в state только цифры

7.2. Альтернативные способы работы с формами

Использование управляемых элементов форм — предпочтительный вариант. Но как вы уже видели, этот способ требует дополнительной работы, потому что вам приходится вручную отслеживать изменения и обновлять состояние. Фактически если вы определяете значение атрибутов `value`, `checked` и `selected` с использованием строк, свойств или состояний, элемент является управляемым (с точки зрения React).

В то же время элементы форм могут быть неуправляемыми, если атрибуты `value` не заданы (ни в состоянии, ни в статическом значении). И хотя это делать не рекомендуется по причинам, перечисленным в начале этой главы (состояние DOM представления может отличаться от внутреннего состояния React), неуправляемые элементы могут быть полезны при построении простой формы, которая будет отправлена серверу. Иначе говоря, возможность применения неуправляемого паттерна может рассматриваться, если только вы не строите сложный UI-элемент с многочисленными изменениями и действиями пользователя; в большинстве случаев так поступать не следует.

Обычно для использования неуправляемых компонентов определяется событие отправки данных формы, как правило, это событие `onClick` для кнопки и/или `onSubmit` для формы. При наличии такого обработчика возможны два варианта:

- Отслеживать изменения так же, как при работе с управляемыми элементами, с использованием состояния для отправки, но не для значений (в конце концов, это же неуправляемое решение!).
- Не отслеживать изменения.

Первый вариант достаточно прост: те же слушатели событий и обновление состояний. Получается слишком большой объем кода, если состояние используется только на последней стадии (для отправки данных формы).

> **ПРЕДУПРЕЖДЕНИЕ** Технология React все еще относительно молода, и оптимальные методы работы формируются на основании практического опыта, связанного не только с написанием, но и с сопровождением приложений. Рекомендации могут измениться на основании нескольких лет сопровождения крупного приложения React. Тема неуправляемых компонентов — неочевидная область, по поводу которой нет четкого консенсуса. Возможно, вы слышали, что это антипаттерн и его следует полностью избегать. Я не хочу принимать чью-либо сторону, но приведу достаточно информации, чтобы вы могли принять собственное решение. Я делаю это, потому что считаю, что вам нужно предоставить всю доступную информацию, а вы достаточно умны, чтобы действовать в соответствии с ней. Мораль: оставшуюся часть этой главы можно считать необязательным материалом — это инструмент, который вы можете (хотя и не обязаны) использовать в своей работе.

7.2.1. Неуправляемые элементы с отслеживанием изменений

Как вам уже известно, в React термин *неуправляемый компонент* означает, что свойство `value` не задается библиотекой React. Когда это происходит, внутреннее значение (или состояние) компонента может отличаться от значения в представлении компонента. Фактически возникает диссонанс между внутренним состоянием и представлением. Состояние компонента может содержать логику (например, проверки данных); с паттерном неуправляемого компонента ваше представление будет получать весь пользовательский ввод в элементе формы, создавая расхождение между представлением и состоянием.

Например, следующее текстовое поле является неуправляемым, потому что React не задает значение:

```
render() {
  return <input type="text" />
}
```

Любой пользовательский ввод будет немедленно рендериться в представлении. Хорошо это или плохо? Наберитесь терпения, мы разберем этот сценарий.

Для отслеживания изменений в неуправляемом компоненте используется событие onChange. Например, у поля ввода на рис. 7.10 имеется обработчик события onChange (`this.handleChange`), ссылка (`textbook`) и текст-подсказка, который выводится серым шрифтом в пустом поле.

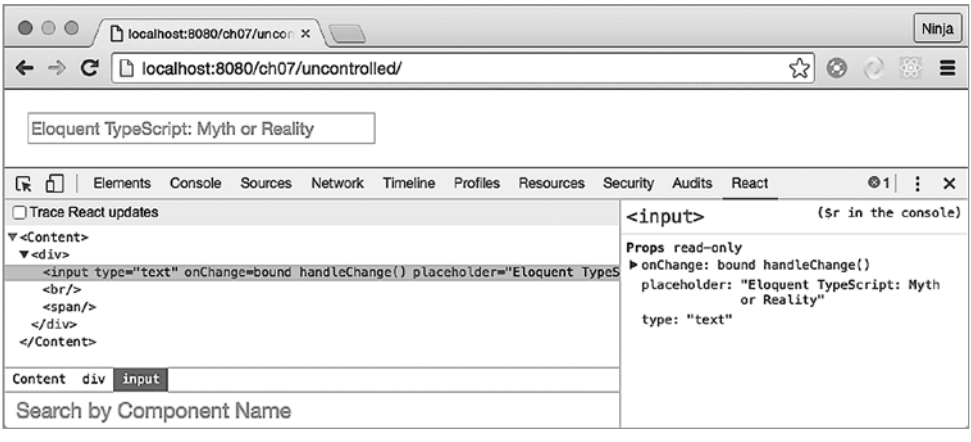


Рис. 7.10. У неуправляемого компонента нет значения, которое задается приложением

В листинге 7.7 представлен метод `handleChange()`, который выводит значения в консоль и обновляет состояние с использованием `event.target.value` (ch07/uncontrolled/jsx/content.jsx).

Листинг 7.7. Неуправляемый элемент с отслеживанием изменений

```
class Content extends React.Component {
  constructor(props){
    super(props)
    this.state = {textbook: ''}
  }
  handleChange(event) {
    console.log(event.target.value)
    this.setState({textbook: event.target.value})
  }
  render() {
    return <div>
      <input
        type="text"
        onChange={this.handleChange}
        placeholder="Eloquent TypeScript: Myth or Reality" />
      <br/>
      <span>{this.state.textbook}</span>
    </div>
  }
}
```

Присваивает пустую строку в качестве исходного значения

Обновляет состояние при каждом изменении в поле ввода

Не задает значение, только слушатель события

Использует для вывода переменной состояния, значение которой будет задаваться в методе handleChange()

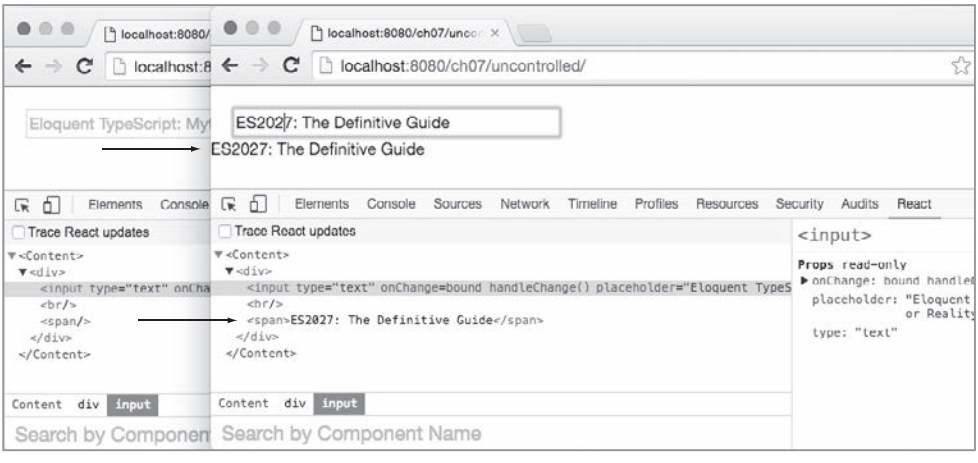


Рис. 7.11. Ввод данных обновляет состояние из-за отслеживания изменений, но значение элемента текстового поля в DOM остается неуправляемым

Суть в том, что пользователь может ввести любые данные, потому что React не управляет значением текстового поля. React только отслеживает новые значения (`onChange`) и задает состояние. В свою очередь, изменение состояния обновляет `<span>` (рис. 7.11).

В этом варианте реализуется обработчик события для поля ввода. Можно ли полностью обойтись без отслеживания изменений?

7.2.2. Неуправляемые элементы без отслеживания изменений

Рассмотрим второй вариант. В нем появляется проблема с готовностью всех значений на тот момент, когда вы хотите их использовать (например, при отправке данных формы). В варианте с отслеживанием изменений все данные доступны в состояниях. Когда вы решаете не отслеживать изменения в неуправляемых элементах, данные все равно остаются в DOM. Чтобы получить данные в объекте JavaScript, следует использовать ссылки, как на рис. 7.12. Сравните с тем, как работают неуправляемые элементы на рис. 7.12, со схемой работы управляемых элементов на рис. 7.1.

> **ПРИМЕЧАНИЕ** При работе с управляемыми компонентами или с неуправляемыми компонентами, сохраняющими данные, данные все время находятся в состоянии. К методу, рассмотренному здесь, это не относится.

Подведем итог: чтобы решение с использованием неуправляемых элементов без отслеживания изменений работало, вам понадобится способ обращения к другим элементам для получения данных от них.

Рис. 7.12. Использование неуправляемого элемента без отслеживания изменений (вместо чего выполняется обращение к значениям по ссылке)

7.2.3. Использование ссылок для обращения к переменным

При работе с неуправляемыми компонентами, не отслеживающими события (такие, как `onChange`), для обращения к значениям используются ссылки, однако ссылки не являются исключительной особенностью этого конкретного паттерна. Вы можете использовать ссылки в любом другом сценарии так, как считаете нужным, хотя использование ссылок обычно осуждается как антипаттерн. Это объясняется тем, что при правильном определении элементов React, когда каждый элемент с внутренним состоянием синхронизируется с состоянием представления (DOM), необходимость в ссылках практически отсутствует. Но понимать, как работают ссылки, все же необходимо, поэтому я расскажу о них в этом разделе.

Ссылки позволяют вам получить элемент DOM (или узел) компонента React.js. Это может быть удобно, когда вам нужно получить значения элементов форм, но не отслеживать изменения в элементах.

Чтобы использовать ссылку, вы должны:

- Убедиться в том, что у элемента в `return` метода `render` имеется атрибут с именем в верблюжьем регистре (например, `email`: `<input ref="userEmail" />`).
- Обратиться к экземпляру DOM по именованной ссылке из другого метода. Например, в обработчике события `this.refs.NAME` превращается в `this.refs.userEmail`.

`this.refs.NAME` дает экземпляр компонента React, но как получить значение? От узла DOM было бы больше пользы! Чтобы обратиться к узлу DOM компонента, вызовите `ReactDOM.findDOMNode(this.refs.NAME)`:

```
let emailNode = ReactDOM.findDOMNode(this.refs.email)
let email = emailNode.value
```

На мой взгляд, запись получается слишком длинной и громоздкой. С учетом этого обстоятельства можно воспользоваться более удобным именем:

```
let fD = ReactDOM.findDOMNode
let email = fD(this.refs.email).value
```

Рассмотрим пример на рис. 7.13, где в форму вводится адрес электронной почты пользователя и комментарий. Значения выводятся на консоль браузера.

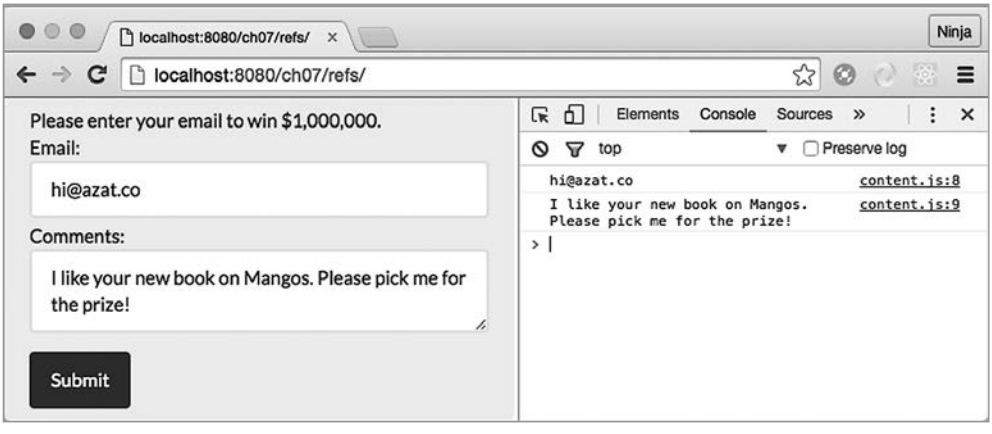


Рис. 7.13. Неуправляемая форма, которая получает данные в двух полях и выводит их на консоль

Структура этого проекта сильно отличается от структур других проектов. Она выглядит так:

```
/email
  /css
    bootstrap.css
  /js
    content.js  <-- Откомпилированный сценарий с главным компонентом
    react.js
    react-dom.js
    script.js
  /jsx
    content.jsx
    script.jsx  <-- Команда ReactDOM.render() в JSX
  index.html
```

Когда нажата кнопка Submit, можно получить доступ к ссылкам `emailAddress` и `comments` и вывести значения на консоль, как показано в листинге 7.8 (ch07/email/jsx/content.jsx).

Листинг 7.8. Начало формы с адресом электронной почты

```
class Content extends React.Component {
  constructor(props) {
    super(props)
    this.submit = this.submit.bind(this)
    this.prompt = 'Please enter your email to win $1,000,000.'
  }
```

Определяет атрибут класса

```
submit(event) {
  let emailAddress = this.refs.emailAddress
  let comments = this.refs.comments
  console.log(ReactDOM.findDOMNode(emailAddress).value)
  console.log(ReactDOM.findDOMNode(comments).value)
}
```

Обращается к значению адреса электронной почты по ссылке

Далее следует обязательная функция `render()`, использующая классы Twitter Bootstrap для стилевого оформления формы (ch07/email/jsx/content.jsx). Не забудьте использовать `className` для атрибута `class`!

Листинг 7.9. Метод render() формы с адресом электронной почты

Выводит значение атрибута prompt компонента Content

Реализует поле для ввода адреса электронной почты с атрибутом элемента placeholder. Свойство placeholder — визуальная подсказка с примером того, какие данные должны вводиться в поле. Поле использует атрибуты элементов className и ref

```
render: function() {
  return (
    <div className="well">
      <p>{this.prompt}</p>
      <div className="form-group">
        Email: <input ref="emailAddress" className="form-control"
        ➥ type="text" placeholder="hi@azat.co"/>
      </div>
      <div className="form-group">
        Comments: <textarea ref="comments" className="form-control"
        ➥ placeholder="I like your website!"/>
      </div>
      <div className="form-group">
         <a className="btn btn-primary" value="Submit"
         ➥ onClick={this.submit}>Submit</a>
      </div>
    </div>
  )
}
})
```

Код кнопки Submit с событием onClick, которое вызывает this.submit

Обычный узел HTML DOM для `<textarea>` использует `innerHTML` в качестве значения. Как упоминалось ранее, в React для этого элемента можно использовать свойство `value`:

```
ReactDOM.findDOMNode(comments).value
```

Дело в том, что React реализует свойство `value`. Это одна из удобных возможностей, которые вы получаете вместе с более последовательным API для элементов форм. В то же время, поскольку метод `ReactDOM.findDOMNode()` возвращает узел DOM, вам доступны другие обычные атрибуты HTML (такие, как `innerHTML`) и методы (как `getAttribute()`).

Теперь вы знаете, как обращаться к элементам и их значениям из практически любого метода компонента, не только из обработчика события этого конкретного

компонента. Еще раз подчеркну, что ссылки нужны только для тех редких случаев, когда вы используете неуправляемые элементы. Злоупотребление ссылками считается нежелательной практикой. В большинстве случаев вам не придется использовать ссылки с управляемыми элементами, потому что вместо них можно использовать состояния компонентов.

Также возможно присвоить функцию атрибуту `ref` в JSX. Эта функция вызывается всего один раз, при подключении элемента. В функции можно сохранить узел DOM в атрибуте экземпляра `this.emailInput`:

```
<input ref={(input) => { this.emailInput = input }}
  className="form-control"
  type="text"
  placeholder="hi@azat.co"/>
```

Неуправляемые компоненты требуют меньшего объема кода (обновления состояния и отслеживание изменений не обязательны), но с ними возникает другая проблема: вы не можете присвоить значения состояниям или жестко фиксированным значениям, потому что тогда они превратятся в управляемые элементы (например, нельзя использовать `value={this.state.email}`). Как задать исходное значение? Допустим, форма в приложении с заявкой на кредит была частично заполнена и сохранена, а пользователь решает продолжить сеанс. Нужно отобразить уже введенную информацию, но вы не можете использовать атрибут `value`. Давайте посмотрим, как задаются значения по умолчанию.

7.2.4. Значения по умолчанию

Предположим, в форме заявки на кредит некоторые поля должны быть заполнены существующими данными. В обычной разметке HTML поле формы определяется с `value`, и пользователи могут изменять элемент на странице. Но React использует `value`, `checked` и `selected` для поддержания соответствия между представлением и внутренним состоянием элементов. В React при использовании жестко фиксированных значений вида

```
<input type="text" name="new-book-title" value="Node: The Best Parts"/>
```

вы получите поле, доступное только для чтения. В большинстве случаев это не то, что требуется. По этой причине в React специальный атрибут `defaultValue` задает значение и позволяет пользователям изменять элементы формы.

Например, представьте, что форма была сохранена ранее и вы хотите заполнить поле `<input>` для удобства пользователя. В этом случае необходимо использовать свойство `defaultValue` для элементов формы. Исходное значение поля задается следующим образом:

```
<input type="text" name="new-book-
    title" defaultValue="Node: The Best Parts"/>
```

Если вместо `defaultValue` используется атрибут `value` (`value="JSX"`), этот элемент становится доступным только для чтения. Он не только будет управляемым, но и его значение не будет изменяться при вводе в элементе `<input>`, как показано на рис. 7.14. Дело в том, что значение жестко фиксировано, и React будет поддерживать это значение. Вероятно, вам хотелось иного. Разумеется, в реальных приложениях значения получаются на программном уровне, что в React означает использование свойств (`this.props.name`):

```
<input type="text" name="new-book-title" defaultValue={this.props.title}/>
```

или состояний:

```
<input type="text" name="new-book-title" defaultValue={this.state.title}/>
```

Рис. 7.14. Значение элемента <input> на веб-странице становится фиксированным (неизменным), когда вы устанавливаете значение в строку

Атрибут React `defaultValue` чаще всего используется с неуправляемыми компонентами; но, как и в случае со ссылками, значения по умолчанию могут использоваться с управляемыми компонентами или в любых других сценариях. Правда, в управляемых компонентах значения по умолчанию уже не настолько актуальны, потому что эти значения можно определить в состоянии в конструкторе: например, `this.state = { defaultName: 'Abe Lincoln'}`.

Как вы уже видели, большая часть работы, связанной с пользовательским интерфейсом, выполняется в элементах форм. Эти элементы должны быть красивыми, но при этом понятными в использовании. А еще вам понадобятся информативные сообщения об ошибках, проверка данных в клиентской части и такие нетривиальные средства, как экранные подсказки (tooltips), масштабируемые переключатели, значения по умолчанию и текст-заполнитель. Построение пользовательского интерфейса может стать сложной задачей, которая быстро выходит из-под контроля! К счастью, React существенно упрощает вашу работу, позволяя использовать межбраузерный API для элементов форм.

7.3. Вопросы

1. Неуправляемый компонент задает значение, а управляемый компонент этого не делает. Да или нет?
2. Какой из следующих вариантов синтаксиса значений по умолчанию является правильным: `default-value`, `defaultValue` или `defVal`?
3. Команда React рекомендует использовать `onChange` вместо `onInput`. Да или нет?

4. Что в следующем списке используется для задания значения `<textarea>`? Дочерние элементы, внутренняя разметка HTML или `value`?
5. К чему в следующем списке относится `selected` в формах: `<input>`, `<textarea>` или `<option>`?
6. Что из перечисленного является лучшим способом получения узла DOM по ссылке: `React.findDomNode(this.refs.email)`, `this.refs.email`, `this.refs.email.getDOMNode`, `ReactDOM.findDOMNode(this.refs.email)` или `this.refs.email.getDomNode`?

7.4. Итоги

- Предпочтительный подход к работе с формами основан на использовании управляемых компонентов со слушателями событий, которые отслеживают и сохраняют данные в состоянии.
- Использование неуправляемых компонентов с отслеживанием элементов или без отслеживания нежелательно, и его следует избегать.
- Ссылки и значения по умолчанию могут использоваться с любыми элементами, но обычно они не нужны для управляемых компонентов.
- Элемент React `<textarea>` использует атрибут `value`, а не внутренний контент.
- `this.refs.NAME` — обращение к ссылкам.
- `defaultValue` позволяет задать исходное значение DOM для элемента.
- `ref="NAME"` используется для определения ссылок.

7.5. Ответы

1. Нет. «Управляемым» называется компонент/элемент, который устанавливает значение.
2. `defaultValue`. Все остальные имена нелегитимны.
3. Да. В обычной разметке HTML `onChange` может и не срабатывать при каждом изменении, но в React это происходит всегда.
4. В React значение задается с использованием `value` ради логической последовательности. Однако в обычной разметке HTML используется внутренняя разметка HTML.
5. `<option>`.
6. Используйте `ReactDOM.findDOMNode(reference)` или обратный вызов (не указывается в числе ответов).

8 Масштабируемость компонентов React

Посмотрите вступительный видеоролик к этой главе, отсканировав QR-код или перейдя по ссылке http://reactquickly.co/videos/ch08.

ЭТА ГЛАВА ОХВАТЫВАЕТ СЛЕДУЮЩИЕ ТЕМЫ:

- Свойства по умолчанию в компонентах.
- Типы свойств React и проверка данных.
- Рендеринг дочерних элементов.
- Создание компонентов более высокого порядка React для повторного использования кода.
- Лучшие практики: презентационные и контейнерные компоненты.

К настоящему моменту вы узнали, как создавать компоненты, как сделать их интерактивными и как работать с пользовательским вводом (событиями и элементами ввода). С этими знаниями вы прошли значительную часть пути к построению сайтов с компонентами React, и все же время от времени неизбежно будут возникать раздражающие проблемы. Это особенно часто происходит в больших проектах, в которых вы полагаетесь на компоненты, созданные другими разработчиками (вашими коллегами или соавторами в проектах с открытым кодом).

Например, если вы используете компонент, написанный кем-то другим, как узнать, предоставляете ли вы правильные свойства для него? И как использовать существующий компонент с небольшой дополнительной функциональностью (которая также применяется к другим компонентам)? Все эти проблемы относятся к области *масштабируемости разработки*, то есть к работе с кодом в процессе разрастания кодовой базы. Некоторые возможности и паттерны React помогут вам в этом.

Эти темы важны, если вы хотите узнать, как эффективно построить сложное приложение React. Например, компоненты более высокого порядка позволяют расширить функциональность компонента, а типы свойств предоставляют безопасность проверки типов и, в определенной степени, — защиту на уровне здравого смысла.

К концу этой главы вы будете знать большинство возможностей React из этой области. Вы узнаете, как сделать ваш код более удобным для разработчика (при помощи типов свойств) и как повысить эффективность вашей работы (при помощи имен компонентов и компонентов более высокого порядка). Возможно, коллеги будут восхищаться элегантностью ваших решений. Все эти возможности помогают более эффективно использовать React, так что перейдем без лишних слов к делу.

ПРИМЕЧАНИЕ Исходный код примеров этой главы доступен по адресам www.manning.com/books/react-quickly и https://github.com/azat-co/react-quickly/tree/master/ch08 (в папке ch08 репозитория GitHub https://github.com/azat-co/react-quickly). Также некоторые демонстрационные примеры доступны по адресу http://reactquickly.co/demos.

8.1. Свойства по умолчанию в компонентах

Представьте, что вы строите компонент `Datepicker`, который получает несколько обязательных свойств, например количество строк, локальный контекст и текущую дату:

```
<Datepicker currentDate={Date()} locale="US" rows={4}/>
```

Что произойдет, если новый участник команды попытается воспользоваться вашим компонентом, но забудет передать необходимое свойство `currentDate`? Или если другой коллега передаст строку `"4"` вместо числа 4? Компонент не сделает ничего (значения не определены) или произойдет нечто похуже — он может вызвать фатальный сбой, и во всем обвинят вас (`ReferenceError`?).

Как ни печально, такая ситуация не так уж редко встречается в веб-разработке, потому что JavaScript относится к числу языков со слабой типизацией. К счастью, React предоставляет возможность определения значений по умолчанию для свойств — статический атрибут `defaultProps`.

Главное преимущество `defaultProps` заключается в том, что при отсутствии свойства будет сгенерировано значение по умолчанию. Чтобы задать значение свойства по умолчанию для класса компонента, определите `defaultProps`. Например, в упомянутом выше определении компонента `Datepicker` можно добавить статический атрибут класса (не атрибут экземпляра, который не сработает, — атрибуты экземпляров задаются в `constructor()`):

```
class Datepicker extends React.Component {
  ...
}
```

```
Datepicker.defaultProps = {
  currentDate: Date(),
  rows: 4,
  locale: 'US'
}
```

Чтобы продемонстрировать применение `defaultProps`, представьте, что у вас имеется компонент, который рендерит кнопку. Обычно кнопки снабжаются надписями, но эти надписи должны настраиваться. Если специальное значение не указано, на кнопке должно выводиться значение по умолчанию.

Надпись кнопки определяется свойством `buttonLabel`, которое используется в атрибуте `return` метода `render()`. Это свойство всегда должно содержать `Submit`, если значение не задано. Для этого вы реализуете статический атрибут класса `defaultProps`, который представляет собой объект со свойством `buttonLabel` и значением по умолчанию:

```
class Button extends React.Component {
  render() {
    return <button className="btn" >{this.props.buttonLabel}</button>
  }
}
Button.defaultProps = {buttonLabel: 'Submit'}
```

Родительский компонент `Content` рендерит четыре кнопки. У трех из этих четырех кнопок отсутствуют свойства:

```
class Content extends React.Component {
  render() {
    return (
      <div>
        <Button buttonLabel="Start"/>
        <Button />
        <Button />
        <Button />
      </div>
    )
  }
}
```

Сможете угадать результат? На первой кнопке выводится надпись Start, а на остальных кнопках — надпись Submit (рис. 8.1).

Определять значения свойств по умолчанию стоит почти всегда, потому что они сделают ваши компоненты более устойчивыми к ошибкам. Другими словами, ваши компоненты становятся умнее, потому что они будут обладать минимальным оформлением и поведением, даже если необходимые данные не были переданы.

Также на происходящее можно взглянуть иначе: определение значения по умолчанию означает, что вам не придется объявлять старое значение снова и снова. Если

вы используете одно значение свойства в большинстве случаев, но при этом хотите оставить возможность изменить это значение (переопределить значение по умолчанию), используйте `defaultProps`. Переопределение значения по умолчанию не создает никаких проблем, как наглядно показывает первая кнопка в этом примере.

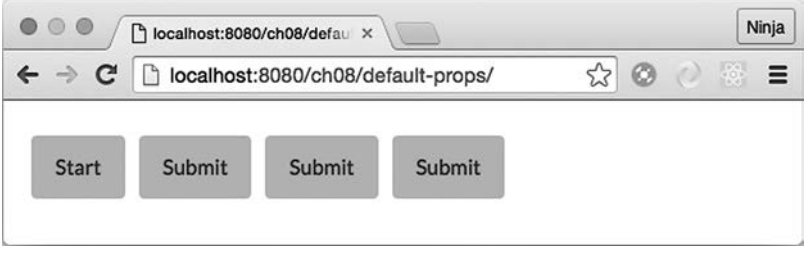


Рис. 8.1. Надпись на первой кнопке была задана при ее создании. У других элементов ее нет, поэтому они используют значение свойства по умолчанию

8.2. Типы свойств React и проверка данных

Возвращаясь к приведенному ранее примеру с компонентом `Datepicker` и коллегами, не знавшими о типах свойств (`"4"` вместо 4), вы можете задать типы свойств, которые должны использоваться с классами компонентов React.js. Для этой цели используется статический атрибут `propTypes`. Он не обеспечивает форсированной проверки типов значений свойств, а всего лишь выдает предупреждение. Таким образом, если вы находитесь в режиме разработки, а тип не совпадает, вы получите предупреждающее сообщение в консоли, и в процессе эксплуатации приложения ничего не будет сделано для того, чтобы помешать использованию неправильного типа. Фактически React.js подавляет это предупреждение в режиме эксплуатации. Таким образом, `propTypes` — в основном вспомогательная функция, которая предупредит вас о несоответствии типов данных на стадии разработки.

REACT В РЕЖИМЕ РАЗРАБОТКИ И ЭКСПЛУАТАЦИИ

Команда React.js определяет режим разработки как режим, использующий не минифицированную версию React (то есть без сжатия), а режим эксплуатации — как режим, использующий минифицированную версию. От авторов React:

«Мы предоставляем две версии React: версию без сжатия для разработки и минифицированную версию для реальной эксплуатации. Версия для разработки включает дополнительные предупреждения о распространенных ошибках, а версия для эксплуатации включает дополнительные оптимизации быстродействия и подавляет все сообщения об ошибках».

Для React 15.5 и более поздних версий (в большинстве примеров из книги используется React v15.5) определения типов предоставляются отдельным пакетом `prop-types` (www.npmjs.com/package/prop-types). Вы должны включить `prop-types` в свой файл HTML. Пакет становится глобальным объектом (`window.PropTypes`):

```
<!-- версия для разработки -->
<script src="https://unpkg.com/prop-types/prop-types.js"></script>
<!-- версия для эксплуатации -->
<script src="https://unpkg.com/prop-types/prop-types.min.js"></script>
```

Если вы используете React 15.4 или более раннюю версию, включать `prop-types` не нужно, потому что типы встроены в React: `React.propTypes`.

Рассмотрим простой пример определения статического атрибута `propTypes` для класса `Datepicker` с разными типами: строковым, числовым и перечисляемым. В этом примере используется React v15.5, а `prop-types` включается в разметку HTML (здесь не показано):

```
class Datepicker extends React.Component {
  ...
}
Datepicker.propTypes = {
  currentDate: PropTypes.string,
  rows: PropTypes.number,
  locale: PropTypes.oneOf(['US', 'CA', 'MX', 'EU'])
}
```

window.PropTypes, потому что сценарий включает prop-types.js

> **ПРЕДУПРЕЖДЕНИЕ** Никогда не полагайтесь на проверку вводимых данных в клиентской части, потому что ее легко можно обойти. Используйте ее только для улучшения опыта взаимодействия и выполняйте все проверки на стороне сервера.

Для проверки типов свойств используйте свойство `propTypes` с объектом, содержащим свойства как ключи, а типы как значения. Типы React.js содержатся в объекте `PropTypes`:

- `PropTypes.string`
- `PropTypes.string`
- `PropTypes.number`
- `PropTypes.bool`
- `PropTypes.object`
- `PropTypes.array`
- `PropTypes.func`
- `PropTypes.shape`
- `PropTypes.any.isRequired`

- `PropTypes.objectOf(PropTypes.number)`
- `PropTypes.arrayOf(PropTypes.number)`
- `PropTypes.node`
- `PropTypes.instanceOf(Message)`
- `PropTypes.element`
- `PropTypes.oneOfType([PropTypes.number, ...])`

Для демонстрации расширим пример `defaultProps`, добавив несколько типов свойств в дополнение к значениям свойств по умолчанию. Этот проект имеет аналогичную структуру: content.jsx, button.jsx и script.jsx. Файл index.html содержит ссылку на prop-types.js:

```
<!DOCTYPE html>
<html>

  <head>
    <script src="js/react.js"></script>
    <script src="js/prop-types.js"></script>
    <script src="js/react-dom.js"></script>
    <link href="css/bootstrap.css" type="text/css" rel="stylesheet"/>
    <link href="css/style.css" type="text/css" rel="stylesheet"/>

  </head>

  <body>
    <div id="content" class="container"></div>
    <script src="js/button.js"></script>
    <script src="js/content.js"></script>
    <script src="js/script.js"></script>
  </body>

</html>
```

Определим класс `Button` с необязательным текстом строкового типа. Чтобы реализовать его, следует определить статический атрибут класса (свойство этого класса) `propTypes` с ключом `title` и значением `PropTypes.string`. Этот код сохраняется в файле button.js:

```
Button.propTypes = {
  title: PropTypes.string
}
```

Также свойство можно объявить обязательным. Для этого добавьте к типу `isRequired`. Например, свойство `title` является обязательным и относится к строковому типу:

```
Button.propTypes = {
  title: PropTypes.string.isRequired
}
```

Кнопка также требует определения свойства `handler`, значением которого является функция (в конце концов, от кнопок без действия особой пользы не было):

```
Button.propTypes = {
  handler: PropTypes.func.isRequired
}
```

Здесь удобно то, что вы можете определить собственную процедуру проверки данных. Чтобы реализовать нестандартную проверку, достаточно создать выражение, которое возвращает экземпляр `Error`, а затем использовать это выражение в `propTypes: {..}` как значение свойства. Например, следующий код проверяет свойство `email` с использованием регулярного выражения из `emailRegularExpression` (которое я скопировал из интернета — а значит, оно должно быть правильным, верно[1]?):

```
...
propTypes = {
    email: function(props, propName, componentName) {
      var emailRegularExpression =
    /^([\w-]+(?:\.[\w-]+)*)@((?:[\w-]+\.)*\w[\w-]{0,66})\.([a-z]{2,6}(?:\.
     ➥ [a-z]{2})?)$/i
      if (!emailRegularExpression.test(props[propName])) {
        return new Error('Email validation failed!')
      }
    }
}
...
```

А теперь соберем все воедино. Компонент `Button` будет вызываться со свойством `title` (строка) и без него, а также со свойством `handler` (необходимая функция). В листинге 8.1 (ch08/prop-types) типы свойств помогают проверить, что `hander` является функцией, `title` — строкой, а `email` соответствует переданному регулярному выражению.

Листинг 8.1. Использование propTypes и defaultProps

```
class Button extends React.Component {
  render() {
    return <button className="btn">{this.props.buttonLabel}</button>
  }
}

Button.defaultProps = {buttonLabel: 'Submit'}

Button.propTypes = {
  handler: PropTypes.func.isRequired,
```

Требует свойства handler со значением функции

[1] Существует много версий регулярного выражения для адресов электронной почты в зависимости от строгости проверки, доменных зон и других критериев. См. «Email Address Regular Expression That 99,99 % Works», http://emailregex.com; «Validate email address in JavaScript?» (вопрос на сайте Stack Overflow), http://mng.bz/zm37; и Regular Expression Library, http://regexlib.com/Search.aspx?k=email.

```
  title: PropTypes.string,
  email(props, propName, componentName) {
    let emailRegularExpression =
    /^([\w-]+(?:\.[\w-]+)*)@((?:[\w-]+\.)*\w[\w-]{0,66})\.([a-z]{2,6}(?:\.
    ➥ [a-z]{2})?)$/i
    if (!emailRegularExpression.test(props[propName])) {
      return new Error('Email validation failed!')
    }
  }
}
```

Определяет проверку адреса электронной почты по регулярному выражению

Определяет необязательное свойство title со строковым значением

Теперь реализуем родительский компонент `Content`, который рендерит шесть кнопок для тестирования предупреждений, генерируемых для типов свойств (ch08/prop-types/jsx/content.jsx).

Листинг 8.2. Рендеринг шести кнопок

```
class Content extends React.Component {
  render() {
    let number = 1
    return (
      <div>
        <Button buttonLabel="Start"/>
        <Button />
        <Button title={number}/>
        <Button />
        <Button email="not-a-valid-email"/>
        <Button email="hi@azat.co"/>
      </div>
    )
  }
}
```

Выдает предупреждение о том, что обработчик отсутствует

Выдает предупреждение о том, что свойство title должно быть строкой

Выдает предупреждение о неправильном формате электронной почты

При выполнении этого кода в консоли выводятся три предупреждения (не забудьте открыть консоль): мои результаты показаны ниже и на рис. 8.2. Первое предупреждение относится к функции `handler`, которая обязательно должна быть задана и которая была опущена для нескольких кнопок:

```
Warning: Failed propType: Required prop `handler` was not specified in
  `Button`. Check the render method of `Content`.
```

Во втором предупреждении говорится о неверном формате электронной почты для четвертой кнопки:

```
Warning: Failed propType: Email validation failed! Check the render method
of `Content`.
```

Третье предупреждение относится к неверному типу значения `title`, которое должно быть строкой (я передал число для одной кнопки):

```
Warning: Failed propType: Invalid prop `title` of type `number` supplied to
`Button`, expected `string`. Check the render method of `Content`.
```

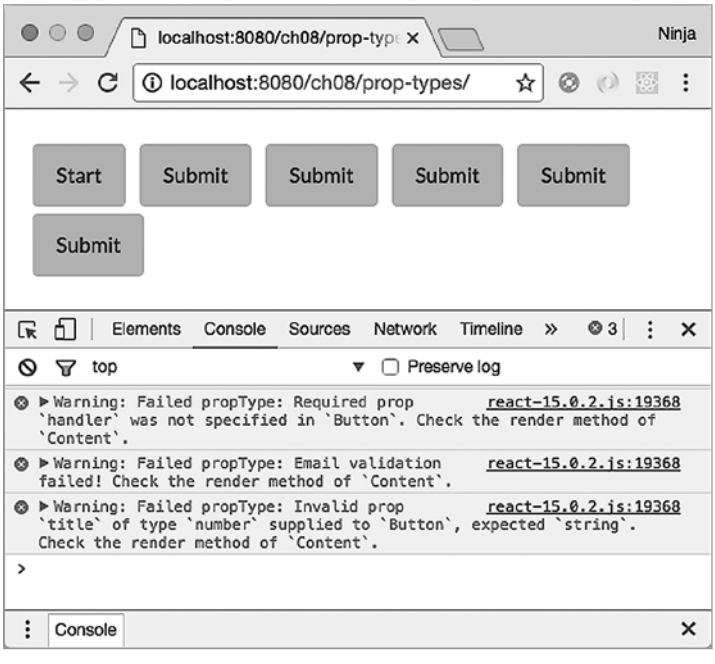


Рис. 8.2. Предупреждения, обусловленные неверным типом свойств

Интересно, что `handler` отсутствует у нескольких кнопок, но вы видите только одно предупреждение. React предупреждает о каждом свойстве только один раз на каждый вызов `render()` для `Content`.

Здесь мне нравится то, что React сообщает, какой родительский компонент следует проверить (`Content` в данном примере). Представьте, что у вас сотни компонентов. Очень полезная информация!

Развернув сообщение в DevTools, вы увидите номер строки элемента `Button`, в которой возникли проблемы и которая привела к выдаче предупреждения. На рис. 8.3 я сначала раскрыл сообщение, а затем нашел свой файл (content.js). В сообщении сказано, что проблема возникла в строке 9.

Если теперь щелкнуть на фрагменте `content.js:9` на консоли, эта строка открывается на вкладке Source (рис. 8.4). Она четко показывает, кто виноват:

```
React.createElement(Button, { title: number }),
```

Вам не понадобятся карты исходного кода (хотя в части 2 вы научитесь создавать и использовать их), чтобы понять, что проблема возникла из-за третьей кнопки.

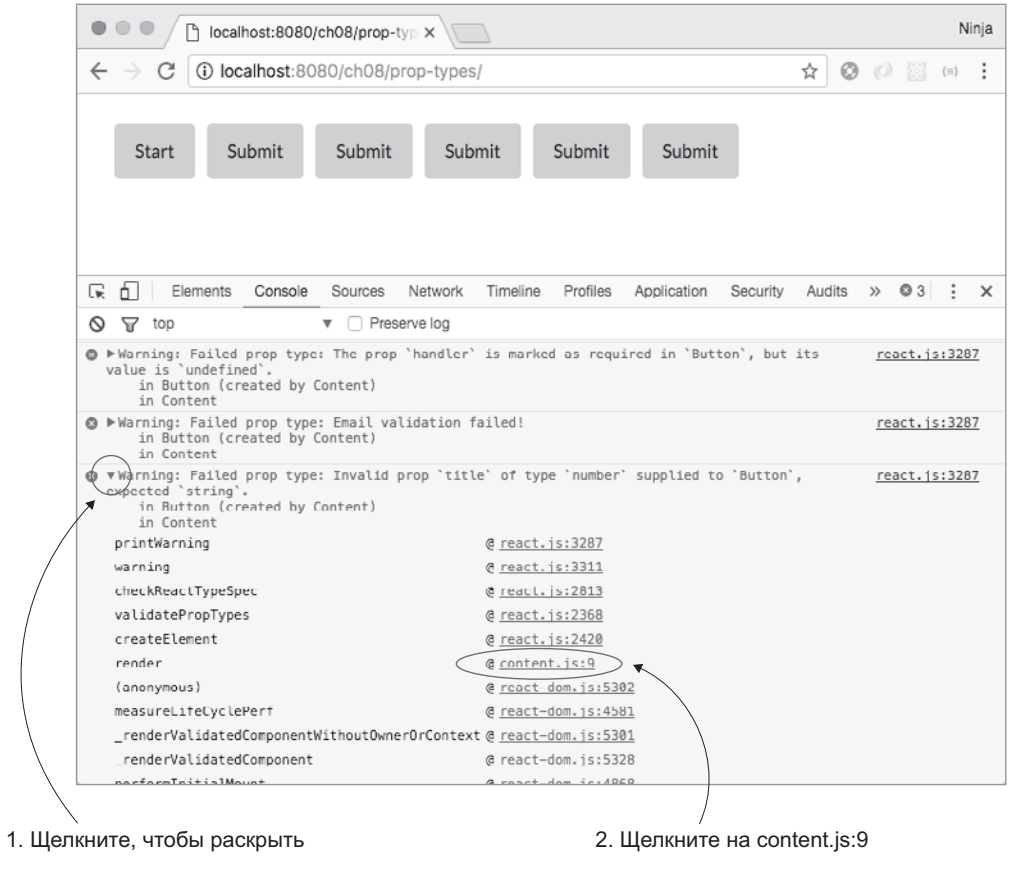


Рис. 8.3. При раскрытии предупреждения виден номер проблемной строки: 9

ПРИМЕЧАНИЕ Еще раз повторю: эти предупреждения выводятся только в неминифицированной версии React (то есть в режиме разработки).

Попробуйте поэкспериментировать с типами свойств и проверкой данных — это удобный механизм. Учтите, что в этом коде используется тот же компонент `Button`:

```
<Button title={number}/>
```

Сможете ли вы обнаружить проблему? Как вы думаете, сколько предупреждений вы получите? (Подсказка: свойства `handler` и `title`.)

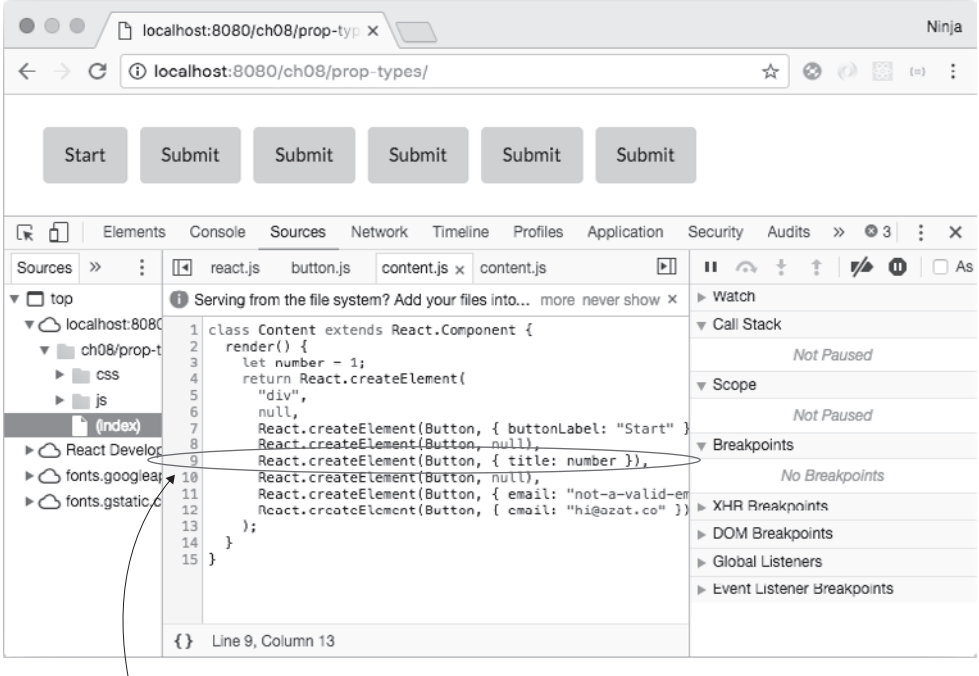


Рис. 8.4. Часто просмотра исходного кода оказывается достаточно для понимания проблемы

КАРТЫ ИСХОДНОГО КОДА

Предупреждения на рис. 8.2 выводятся из-за плохо написанного компонента Content (я написал его так специально — чтобы показать, как работают атрибуты `defaultProps` и `propTypes`). Предупреждения указывают, в каком компоненте и в какой строке компонента возникла проблема.

Однако номера строк не будут соответствовать вашему исходному коду, потому что они относятся к откомпилированному коду JavaScript, а не к JSX. Чтобы получить правильные номера строк, необходимо воспользоваться плагином создания карт исходного кода, например `source-map-support` (https://github.com/evanw/node-source-map-support) или Webpack. В главе 12 рассматривается Webpack.

Для поддержки карт исходного кода также можно воспользоваться Babel: добавьте `--sourceMaps=true` в команду и/или в сценарий сборки package.json. За дополнительной информацией о Babel обращайтесь по адресу https://babeljs.io/docs/usage/options/#options.

Важно знать и использовать `propTypes` (типы свойств и нестандартная проверка данных) в больших проектах или компонентах с открытым кодом. Конечно, типы свойств не обеспечивают жесткого контроля типов и не выдают исключений, но при использовании компонента, написанного другими разработчиками, вы по крайней мере сможете убедиться в том, что переданные свойства относятся к правильному типу. Это относится и к использованию ваших компонентов другими разработчиками. Они оценят тот факт, что вы указали правильные типы свойств. Результат — более комфортная разработка для всех!

Наконец, существует много других типов и вспомогательных методов. За полным списком обращайтесь к документации по адресу http://mng.bz/4Lep.

8.3. Рендеринг дочерних элементов

Продолжим работу над вымышленным проектом React; вместо `Datepicker` (который теперь стал достаточно надежным и предупреждает вас обо всех отсутствующих или неправильно заданных свойствах) вам теперь поручено создать компонент, обладающий достаточной универсальностью для использования любых переданных ему дочерних элементов. Это компонент сообщения в блоге `Content`, который может состоять из заголовка и абзаца текста:

```
<Content>
  <h1>React.js</h1>
  <p>Rocks</p>
</Content>
```

Другое сообщение в блоге может содержать изображение (взять хотя бы Instagram или Tumblr):

```
<Content>
  <img src="images/azat.jpg" width="100"/>
</Content>
```

В обоих сообщениях используется компонент `Content`, но ему передаются разные дочерние элементы. Разве не здорово было бы иметь специальный способ рендера любых дочерних элементов (будь то `<p>` или `<img>`)? Такой способ существует.

Свойство `children` предоставляет простую возможность для ренедера всех дочерних элементов конструкцией `{this.props.children}`. Впрочем, ваши возможности даже не ограничиваются одним рендерингом. Например, можно добавить `<div>` и передать с дочерними элементами:

```
class Content extends React.Component {
  render() {
    return (
      <div className="content">
        {this.props.children}
```

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

Родитель `Content` содержит дочерние элементы `<h1>` и `<p>`:

```
ReactDOM.render(
  <div>
    <Content>
      <h1>React</h1>
      <p>Rocks</p>
    </Content>
  </div>,
  document.getElementById('content')
)
```

В результате `<h1>` и `<p>` заключаются в контейнер `<div>` с классом `content` (рис. 8.5). Помните: для атрибутов класса в React используется `className`.

Разумеется, к такому компоненту, как `Content`, можно добавить еще много всего, например новые классы для стилевого оформления и макеты, и даже использовать

Рис. 8.5. Рендеринг компонента Content с заголовком и абзацем текста с использованием this.props.children

свойства и интерактивность с событиями и состояниями. Благодаря `this.props.children` вы можете создавать компоненты, обладающие гибкостью, мощью и универсальностью.

Допустим, помимо текста и графики необходимо вывести ссылку или кнопку как в предыдущем примере. Компонент `Content` по-прежнему останется оберткой-`<div>` с классом CSS `content` (свойство `className`), но теперь дочерних элементов будет больше. Преимущество такого решения в том, что `Content` может не располагать информацией о своих дочерних элементах — вам не нужно вносить изменения в класс `Content`.

Поместите дочерние элементы в `Content` при создании экземпляра класса (ch08/children/jsx/script.jsx).

Листинг 8.3. Рендеринг элементов с использованием Content

```
ReactDOM.render(
  <div>
    <Content>
      <h1>React</h1>
      <p>Rocks</p>
    </Content>
    <Content>
      <img src="images/azat.jpg" width="100"/>
    </Content>
    <Content>
      <a href="http://react.rocks">http://react.rocks</a>
    </Content>
    <Content>
      <a className="btn btn-danger"
      ➥ href="http://react.rocks">http://react.rocks</a>
    </Content>
  </div>,
  document.getElementById('content')
)
```

Полученная разметка HTML содержит два элемента `<div>` с классами CSS `content`. Один элемент содержит `<h1>` и `<p>`, а другой — `<span>`, как показано в DevTools на рис. 8.6.

В свойстве `children` интересно еще и то, что оно может быть массивом при наличии нескольких дочерних элементов (рис. 8.5). Для обращения к отдельным элементам используется запись следующего вида:

```
{this.props.children[0]}
{this.props.children[1]}
```

Будьте осторожны с проверкой дочерних элементов. Если дочерний элемент только один, `this.props.children` не является массивом. Если вы используете `this.props.children.length`, а единственный дочерний узел является строкой, это может

привести к ошибкам, потому что `length` является действительным свойством для строк. Вместо этого для получения точного количества дочерних элементов следует использовать `React.Children.count(this.props.children)`.

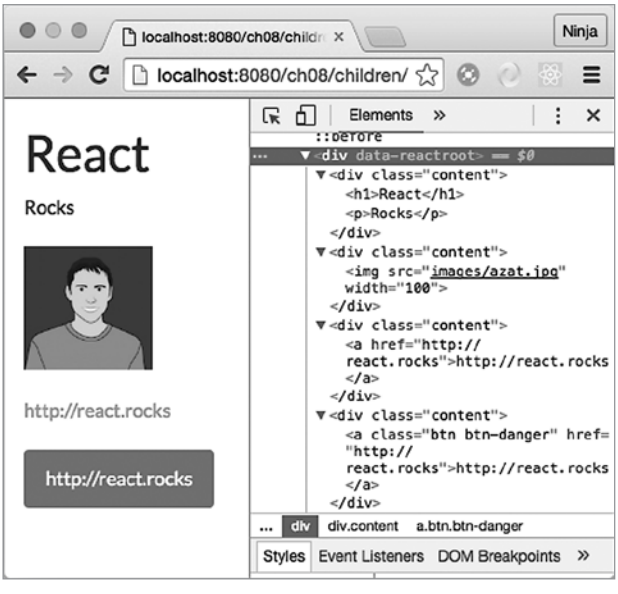


Рис. 8.6. Рендеринг четырех элементов с разным контентом с использованием одного класса компонента

Кроме `React.Children.count`, в React также существуют другие вспомогательные методы. Самыми интересными из них (на мой взгляд) являются следующие:

- `React.Children.map()`
- `React.Children.forEach()`
- `React.Children.toArray()`

Повторять здесь постоянно изменяющийся список было бы неразумно; официальная документация доступна по адресу http://mng.bz/Oi2W.

8.4. Создание компонентов React высшего порядка для повторного использования кода

Как и прежде, будем считать, что вы работаете в большой команде и создаете компоненты, используемые другими разработчиками в их проектах. Допустим, вы работаете над блоком интерфейса. Трое ваших коллег просят вас реализовать

вариант для загрузки ресурса, но каждый из них хочет использовать собственное визуальное представление для кнопки, изображения и ссылки. Вероятно, можно реализовать метод и вызвать его из обработчика события, но существует и более элегантное решение: *компоненты высшего порядка*.

Компонент высшего порядка (HOC, Higher-Order Component) позволяет расширить компонент дополнительной логикой (рис. 8.7). Этот паттерн можно рассматривать как наследование функциональности компонентами при использовании с HOC. Другими словами, HOC позволяют повторно использовать код. Так вы и ваша команда сможете совместно использовать функциональность между компонентами React.js. Тем самым вы избегаете повторений (принцип DRY, http://mng.bz/1K5k).

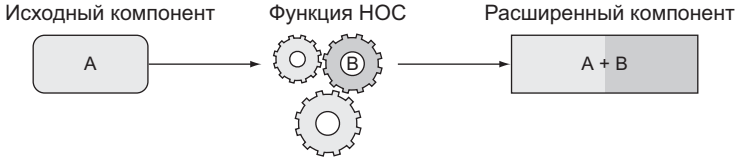


Рис. 8.7. Упрощенная схема паттерна компонента высшего порядка: расширенный компонент обладает свойствами не только A, но A и B

По сути, HOC представляют собой классы компонентов React, которые рендерят исходные классы, попутно добавляя к ним некую дополнительную функциональность. HOC определяются достаточно просто, потому что это всего лишь функции. Для их объявления используется «стрелочный» синтаксис:

```
const LoadWebsite = (Component) => {
  ...
}
```

Имя `LoadWebsite` выбрано произвольно; вы можете присвоить HOC все что угодно при условии, что вы используете то же имя при расширении компонента. Это относится к аргументу функции (`LoadWebsite`): это исходный (то есть еще не расширенный) компонент.

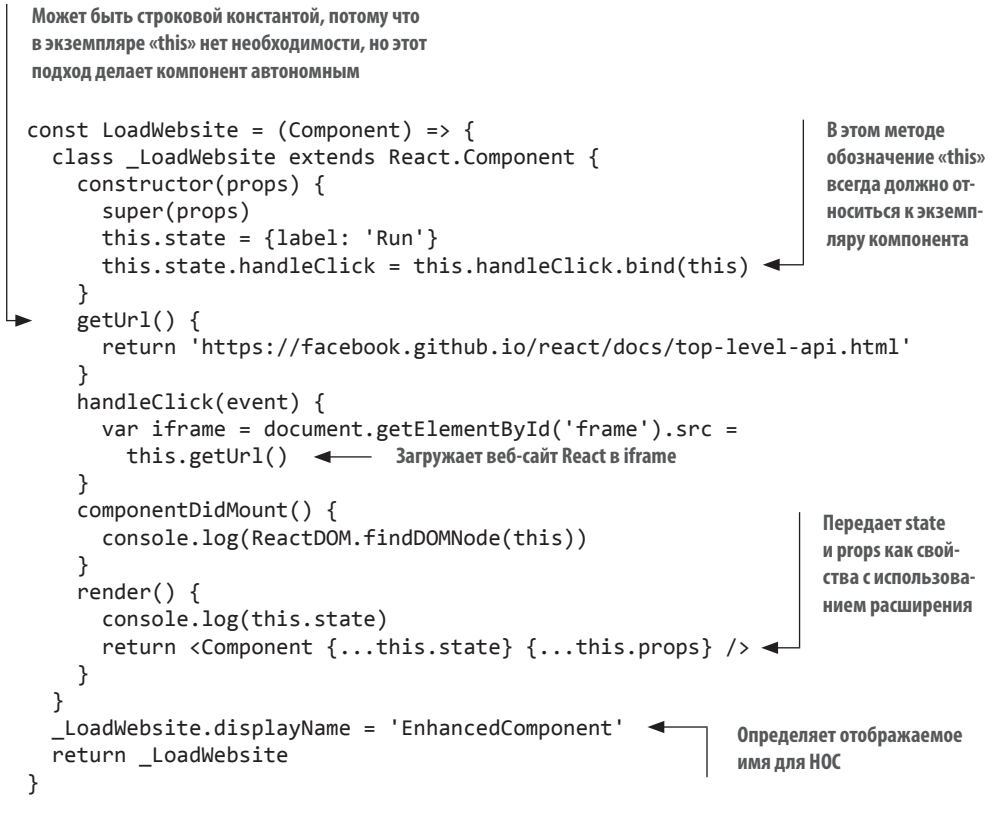
Для наглядности создадим проект для трех ваших коллег. Проект имеет структуру, описанную ниже: три компонента без состояния — `Button`, `Link` и `Logo` в elements.jsx и функцию HOC в load-website.jsx:

```
/hi-order
  /css
    bootstrap.css
    style.css
  /js
    content.js
    elements.js
    load-website.js
```

```
    react.js
    react-dom.js
    script.js
  /jsx
    content.jsx
    elements.jsx
    load-website.jsx
    script.jsx
  index.html
  logo.png
```

Вашим коллегам нужен текст на кнопке и обработчик события щелчка. Давайте зададим надпись и определим метод `handleClick()`. События подключения отмечают жизненный цикл компонента (ch08/hi-order/jsx/load-website.jsx).

Листинг 8.4. Реализация компонента высшего порядка



Ничего сложного, верно? Здесь встречаются два приема, которые еще не использовались в этой книге: `displayName` и оператор расширения `...`. Давайте быстро рассмотрим их.

8.4.1. Использование displayName: как отличить дочерние компоненты от родителя

По умолчанию JSX использует имя класса как имя экземпляра (элемента). Таким образом, элементам, созданным с HOC из этого примера, присваиваются имена `_LoadWebsite`.

Если вы захотите изменить это имя, воспользуйтесь статическим атрибутом `displayName`. Как вам, вероятно, известно, статические атрибуты классов в ES6 должны определяться вне определения класса. (На момент написания книги стандарт для статических атрибутов еще не был завершен.)

Подведем итог: `displayName` необходимо для того, чтобы задать имена элементов React там, где они должны отличаться от имени класса компонента (рис. 8.8). Вы увидите, насколько удобно использовать `displayName` в HOC load-website.jsx для расширения имени, потому что по умолчанию имя компонента совпадает с именем функции (которое не всегда оказывается именно тем, что вам нужно).

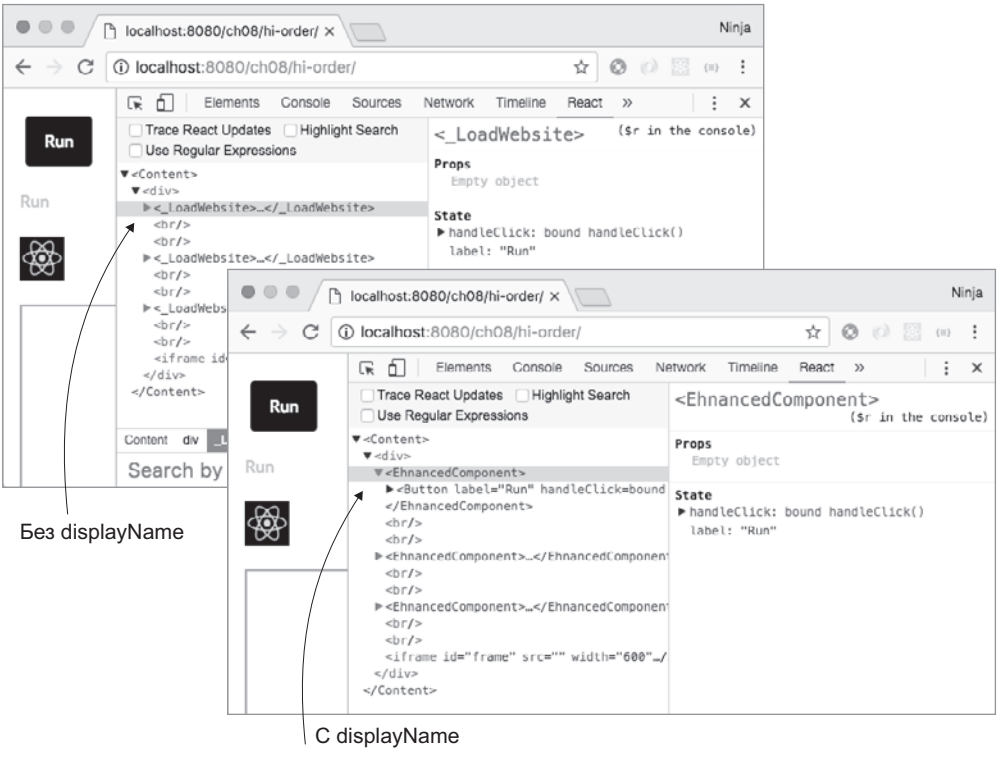


Рис. 8.8. При помощи статического атрибута displayName можно заменить имя компонента _LoadWebsite на EnhancedComponent

ПОДЧЕРКИВАНИЕ В JAVASCRIPT

В JavaScript символ подчеркивания (_) является действительным символом для имени (он используется в библиотеках Lodash и Underscore). Кроме того, символ подчеркивания в начале имени переменной или метода означает, что это приватный атрибут, переменная или метод, не предназначенные для использования в общедоступном интерфейсе (например, из другого модуля, класса, объекта, функции и т. д.). Использовать приватные API в высшей степени не рекомендуется, потому что они с большой вероятностью могут измениться или содержать недокументированное поведение.

Символ подчеркивания в начале имени — соглашение, а не правило; это означает, что оно не обеспечивается исполнительным ядром или платформой. Это всего лишь распространенный паттерн, признанный разработчиками JavaScript. Другими словами, методы и переменные не становятся приватными автоматически, если в их именах использован символ _. Чтобы сделать переменную или метод приватными, используйте замыкание (см. http://developer.mozilla.org/en/docs/Web/JavaScript/Closures и http://javascript.crockford.com/private.html).

8.4.2. Использование оператора расширения: передача всех атрибутов

Теперь рассмотрим оператор расширения (`...`). Он является частью спецификации ES6+/ES2015+ для массивов (http://mng.bz/8fjN); на момент написания книги было выдвинуто предложение использовать расширения для объектов (https://github.com/sebmarkbage/ecmascript-rest-spread). Вполне естественно, что команда React добавила поддержку расширений в JSX.

Идея достаточно проста. Оператор расширения позволяет передать все атрибуты объекта (`obj`) как свойства при использовании в элементе:

```
<Component {...obj}/>
```

Мы использовали расширение в load-website.jsx для передачи переменных свойств и состояния исходному компоненту при его рендере. Это было необходимо, потому что количество свойств, передаваемых функции в аргументах, было заранее неизвестно; таким образом, оператор расширения является обобщенной конструкцией для передачи всех данных (в переменной или объекте).

В React и JSX можно использовать несколько операторов расширения или смешать их с традиционными объявлениями свойств *ключ=значение*. Например, вы можете передать все состояния и все свойства из текущего класса, а также `className` новому элементу `Component`:

```
<Component {...this.state} {...this.props} className="main" />
```

Рассмотрим пример с дочерними элементами. В этой ситуации использование оператора расширения с `this.props` передаст все свойства `DoneLink` элементу ссылки `<a>`:

```
class DoneLink extends React.Component {
  render() {
    return <a {...this.props}>
      <span class="glyphicons glyphicons-check"></span>
      {this.props.children}
      </a>
  }
}
ReactDOM.render(
  <DoneLink href="/checked.html">
    Click here!
  </DoneLink>,
  document.getElementById('done-link')
)
```

Получает любые свойства, переданные DoneLink, и копирует их в <a>

Использует Glyphicons (http://glyphicons.com) для вывода значка пометки

Передает значение для href

В HOC все свойства и состояния передаются исходному компоненту при его рендере. При этом вам не нужно вручную добавлять свойства или удалять их из `render()` каждый раз, когда вы захотите передать что-то новое или прекратить передачу существующих данных из `Content`, где вы создаете экземпляр `LoadWebsite`/`EnhancedComponent` для каждого исходного элемента.

8.4.3. Использование компонентов высшего порядка

Вы узнали, как использовать `displayName` и `...` в JSX и React. А теперь посмотрим, как использовать HOC.

Вернемся к `Content` и файлу content.jsx, где использовался компонент `LoadWebsite`. После определения HOC необходимо создать компоненты, использующие его, в content.jsx:

```
const EnhancedButton = LoadWebsite(Button)
const EnhancedLink = LoadWebsite(Link)
const EnhancedLogo = LoadWebsite(Logo)
```

Затем мы реализуем три компонента — `Button`, `Link` и `Logo` — для повторного использования кода в паттерне HOC. Компонент `Button` создается через `LoadWebsite` и в результате волшебным образом наследует его свойства (`this.props.handleClick` и `this.props.label`):

```
class Button extends React.Component {
  render() {
    return <button
      className="btn btn-primary"
      onClick={this.props.handleClick}>
      {this.props.label}
    </button>
  }
}
```

Компонент `Link` создается HOC, что позволяет вам использовать свойства `handleClick` и `label`:

```
class Link extends React.Component {
  render() {
    return <a onClick={this.props.handleClick} href="#">
     ➥ {this.props.label}</a>
  }
}
```

И наконец, компонент `Logo` использует те же свойства. Да, всё верно: они оказались здесь, потому что вы использовали оператор расширения при создании `Logo` в content.jsx:

```
class Logo extends React.Component {
  render() {
    return <img onClick={this.props.handleClick} width="40" src="logo.png"
     ➥ href="#"/>
  }
}
```

Три компонента рендерятся по-разному, но все они получают `this.props.handleClick` и `this.props.label` из `LoadWebsite`. Рендеринг элементов родительским компонентом `Content` продемонстрирован в листинге 8.5 (ch08/hi-order/jsx/content.jsx).

Листинг 8.5. Совместное использование обработчика события HOC

```
const EnhancedButton = LoadWebsite(Button)
const EnhancedLink = LoadWebsite(Link)
const EnhancedLogo = LoadWebsite(Logo)

class Content extends React.Component {
  render() {
    return (
      <div>
        <EnhancedButton />
        <br />
        <br />
        <EnhancedLink />
        <br />
        <br />
        <EnhancedLogo />
        <br />
        <br />
        <iframe id="frame" src="" width="600" height="500"/>
      </div>
    )
  }
}
```

Объявляет элемент iframe, в котором метод click загружает сайт React

Наконец, не забудьте выполнить рендер `Content` в последних строках script.jsx:

```
ReactDOM.render(
  <Content />,
  document.getElementById('content')
)
```

Когда вы откроете страницу, на ней будут находиться три элемента (`Button`, `Link` и `Logo`). Элементы обладают одинаковой функциональностью: они загружают `IFrame` по щелчку, как показано на рис. 8.9.

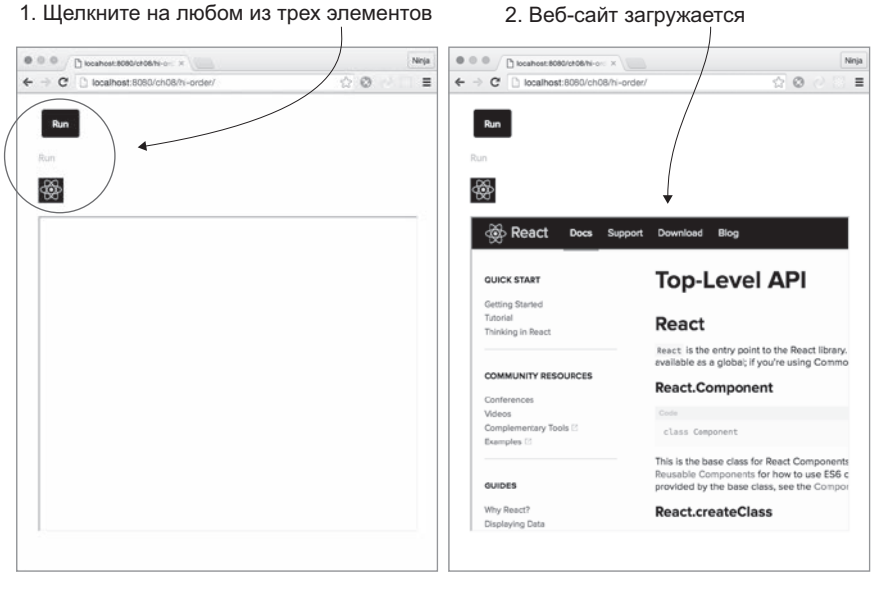


Рис. 8.9. Все три компонента загружают веб-сайт React благодаря функции, предоставившей код его загрузки

Как вы могли убедиться, компоненты HOC отлично подходят для абстрагирования кода. Вы можете использовать их для написания собственных мини-модулей, которые представляют собой компоненты React, пригодные для повторного использования. HOC наряду с типами свойств являются превосходными инструментами для создания удобных компонентов, с которыми будут с удовольствием работать другие разработчики.

8.5. Презентационные и контейнерные компоненты

Существует одно различие, которое позволяет вам масштабировать свой код в соответствии с размерами кода и команды: речь идет о различии между презентаци-

онными и контейнерными компонентами. Мы затронули их в предыдущих главах, но сейчас, поскольку вы уже знаете о передаче дочерних элементов и HOC, нам будет проще говорить о контейнерных компонентах.

Вообще говоря, разбиение кода на два типа упрощает его и делает более удобным для сопровождения. Презентационные компоненты обычно только добавляют структуру в DOM и стилевое оформление. Они получают свойства, но часто не имеют собственных состояний. В большинстве случаев для презентационных компонентов без состояния можно использовать функции. Например, `Logo` является хорошим примером презентационного компонента в стиле класса:

```
class Logo extends React.Component {
  render() {
    return <img onClick={this.props.handleClick} width="40" src="logo.png"
     ➥ href="#"/>
  }
}
```

или в функциональном стиле:

```
const Logo = (props)=>{
  return <img onClick={props.handleClick} width="40" src="logo.png"
  ➥ href="#"/>
}
```

Презентационные компоненты часто используют `this.props.children` в качестве обертки для стилевого оформления дочерних компонентов. Примеры — `Button`, `Content`, `Layout`, `Post` и т. д. С другой стороны, они редко имеют дело с данными и состояниями; это задача контейнерных компонентов.

Контейнерные компоненты часто генерируются компонентами HOC для внедрения источников данных. Они обладают состоянием. Примеры — `SaveButton`, `ImagePostContent` и т. д. Как презентационные, так и контейнерные компоненты могут содержать другие презентационные или контейнерные компоненты; но на первых порах вы обычно будете использовать презентационные компоненты, которые содержат *только* другие презентационные компоненты. Контейнерные компоненты содержат либо другие контейнерные компоненты, либо презентационные компоненты.

Лучше всего начинать с компонентов, которые подходят для ваших целей. Если вы начинаете замечать повторяющиеся закономерности или свойства, передаваемые через несколько уровней вложенных компонентов, но не используемые в промежуточных компонентах, попробуйте ввести один-два контейнерных компонента.

> **ПРИМЕЧАНИЕ** Возможно, вам также попадутся такие термины, как «тощий» (skinny) или «толстый» (fat) компонент. Это синонимы для презентационных и контейнерных компонентов, которые относительно недавно вошли в терминологию React.

8.6. Вопросы

1. React предоставляет мощную поддержку проверки данных, которая снимает необходимость в проверке ввода на стороне сервера. Да или нет?
2. Кроме назначения свойств с использованием `defaultProps`, вы также можете задать их в конструкторе в синтаксисе `this.props.NAME = ЗНАЧЕНИЕ`. Да или нет?
3. Свойство `children` может быть массивом или узлом. Да или нет?
4. Паттерн компонента высшего порядка реализуется функцией. Да или нет?
5. Главное различие между версиями библиотечных файлов React для разработки (неминифицированной) и для эксплуатации (минифицированной) заключается в том, что неминифицированная версия выдает предупреждения, а минифицированная содержит оптимизированный код. Да или нет?

8.7. Итоги

- Для любого свойства компонента можно определить значение по умолчанию, задав атрибут `defaultProps` компонента.
- При работе с несжатой версией библиотеки React (режим разработки) можно включить проверку данных для значений свойств компонентов.
- Вы можете проверить тип свойства, установить атрибут `isRequired` для обязательных свойств или определить собственную нестандартную проверку данных.
- Если значение свойства не проходит проверку, в консоль браузера выводится предупреждение.
- Минифицированная версия библиотеки React не поддерживает эти проверки данных.
- React позволяет инкапсулировать и повторно использовать стандартные свойства, методы и события ваших компонентов посредством создания компонентов высшего порядка.
- Компоненты высшего порядка (HOC) определяются как функции, которые получают другой компонент в аргументе. Аргумент содержит компонент, наследующий от HOC.
- К любым компонентам React или HTML, вложенным в элемент JSX, можно обратиться через свойство `props.children` родительского компонента.

8.8. Ответы

1. Нет. Проверка данных в клиентской части не заменяет проверки в серверной части. Код клиентской части открыт для всех, и кто угодно может обойти его, проанализировав взаимодействие приложения клиентской части с сервером и отправляя данные напрямую серверу.
2. Нет. React требует, чтобы статический атрибут класса `defaultProps` определялся при создании элемента, тогда как `this.props` является атрибутом экземпляра.
3. Да. Если дочерний элемент только один, то `this.props.children` содержит только один узел.
4. Да. Паттерн HOC реализуется функцией, которая получает компонент и создает другой класс компонента с расширенной функциональностью. Этот новый класс рендерит исходный компонент, передавая ему свойства и состояния.
5. Да. Минифицированная версия не выводит предупреждения.

9 Компонент меню

Посмотрите вступительный видеоролик к этой главе, отсканировав QR-код или перейдя по ссылке http://reactquickly.co/videos/ch09.

ЭТА ГЛАВА ОХВАТЫВАЕТ СЛЕДУЮЩИЕ ТЕМЫ:

- Структура проекта и инициализация.
- Построение меню без JSX.
- Построение меню в JSX.

В следующих трех главах мы рассмотрим несколько проектов, постепенно задействующих концепции, представленные в главах 1–8. Эти проекты укрепят материал за счет повторения некоторых приемов и идей, чрезвычайно важных в React. Первый проект очень небольшой, но не пропускайте его.

Представьте, что вы работаете на обобщенном визуальном фреймворке, который будет использоваться во всех приложениях вашей компании. Постоянство внешнего вида и поведения в разных приложениях может быть чрезвычайно важным фактором. Только подумайте, как Twitter Bootstrap используется во многих приложениях Twitter, а Google's Material UI[1] — во многих системах, принадлежащих Google: AdWords, Analytics, Search, Drive, Docs и т. д.

Ваша первая задача — реализовать меню вроде того, что изображено на рис. 9.1. Оно будет использоваться в шапке макета многих страниц разных приложений. Пункты меню должны изменяться в зависимости от роли пользователя и от текущей части приложения. Например, у администраторов и менеджеров в меню должна присутствовать команда управления пользователями. В то же время этот макет должен

[1] Twitter Bootstrap: http://getbootstrap.com. Компоненты React, реализующие Twitter Bootstrap: https://react-bootstrap.github.io. Google Material Design: https://material.io. Компоненты React, реализующие Material Design: www.material-ui.com.

Рис. 9.1. Меню, которое вам предстоит построить

использоваться в приложении для работы с клиентами, которому нужен собственный уникальный набор пунктов меню. В общем, вы поняли: меню должно строиться динамически, а это означает, что команды меню должны генерироваться кодом React.

Для упрощения пункты меню будут реализованы простыми тегами `<a>`. Мы создадим два нестандартных компонента React: `Menu` и `Link`. Это будет сделано примерно так же, как мы создавали компонент `HelloWorld` в главе 1, или так же, как создается любой другой компонент, если уж на то пошло.

Этот проект покажет, как сгенерировать на программном уровне вложенные элементы. Ручное кодирование команд меню не лучшая мысль — а если вам понадобится изменить содержимое меню? Для этой цели следует использовать метод `map()`.

> **ПРИМЕЧАНИЕ** Чтобы повторить приведенное описание работы над проектом, необходимо загрузить неминифицированную версию React (чтобы воспользоваться полезными предупреждениями, которые она возвращает при возникновении каких-либо проблем). Вы также можете загрузить и установить Node.js и npm. Они не являются строго необходимыми для данного проекта, но пригодятся для компиляции JSX позднее в этой главе. Установка обеих программ описана в приложении А.

> **ПРИМЕЧАНИЕ** Исходный код примеров этой главы доступен по адресам www.manning.com/books/react-quickly и https://github.com/azat-co/react-quickly/tree/master/ch09 (в папке ch09 репозитория GitHub https://github.com/azat-co/react-quickly). Также некоторые демонстрационные примеры доступны по адресу http://reactquickly.co/demos.

9.1. Структура проекта и инициализация

Начнем с обзора структуры проекта. Для простоты проект был создан плоским:

```
/menu
  index.html   <-- Главный файл HTML
  package.json
  react-dom.js
  react.js
  script.js    <-- Главный сценарий
```

Учтите, что такой результат вы получите к концу прохождения этого материала, а начинается работа с пустой папки. Итак, создайте новую папку и начните реализацию проекта:

```
$ mkdir menu
$ cd menu
```

Загрузите файлы react.js и react-dom.js версии 15, разместите их в папке. Затем идет файл HTML:

```
<!DOCTYPE html>
<html>
<head>
<script src="react.js"></script>
<script src="react-dom.js"></script>
</head>
```

Разметка HTML для этого проекта самая элементарная. В нее включаются файлы react.js и react-dom.js, которые для простоты находятся в одной папке с файлом HTML. Конечно, позднее файлы *.js следует переместить в другую папку — js, src и т. д.

Тело разметки состоит всего из двух элементов. Первый элемент — контейнер `<div>` с идентификатором `menu`; здесь будет выполняться рендеринг меню. Второй элемент — тег `<script>` с кодом приложения React:

```
<body>
    <div id="menu"></div>
    <script src="script.js"></script>
</body>
</html>
```

С инициализацией проекта покончено. Это тот фундамент, на котором будет построено наше меню — сначала без JSX.

9.2. Построение меню без JSX

script.js — основной файл приложения — содержит код `ReactDOM.render()`, а также два компонента (ch09/menu/script.js).

Листинг 9.1. Заготовка сценария Menu

```
class Menu extends React.Component {...}      <-- Определение Menu

class Link extends React.Component {...}      <-- Определение компонента Link, который используется Menu

ReactDOM.render(
  React.createElement(
    Menu,
    Null      <-- Никакие свойства Menu не передаются
  ),
  document.getElementById('menu')
)
```

Конечно, можно сделать так, чтобы компонент `Menu` зависел от внешнего списка пунктов меню, передаваемого в свойстве (например, `menuOptions`), которое определяется в другом месте:

```
const menuOptions = [...]
//...
ReactDOM.render(
  React.createElement(
    Menu,
    {menus: menuOptions}
  ),
  document.getElementById('menu')
)
```

Оба подхода допустимы. Вам придется выбрать один из них в зависимости от того, как вы ответите на вопрос: хотите ли вы, чтобы компонент `Menu` ограничивался структурой и стилевым оформлением или он также должен обеспечивать получение информации? В этой главе мы используем первый вариант и сделаем компонент `Menu` автономным.

9.2.1. Компонент Menu

А теперь перейдем к созданию компонента `Menu`. Разберем его код шаг за шагом. Чтобы создать его, следует расширить `React.Component()`:

```
class Menu extends React.Component {...}
```

Компонент `Menu` рендерит отдельные пункты меню, которые представляют собой теги ссылок. Но прежде чем отрендерить их (ссылки), необходимо определить сами пункты. Они жестко зафиксированы в массиве `menus` (в более реалистичном приложении эти данные будут получены из модели данных, загружены из хранилища или с сервера):

```
render() {
  let menus = ['Home',  ◄——  Имитация хранилища данных
    'About',
    'Services',
    'Portfolio',
    'Contact us']
    //...
```

Теперь нужно вернуть элементы `Link` для меню (все четыре). Вспомните, что `return` может содержать только один элемент. По этой причине четыре ссылки заключаются в контейнер `<div>`. Начало контейнерного элемента `<div>` без атрибутов выглядит так:

```
return React.createElement('div',
  null,
  //... Ссылки будут сгенерированы позднее
```

Стоит упомянуть, что `{}` может выводить не только переменные и выражения, но и массивы. Это может быть удобно при работе со списками. Чтобы вывести все элементы массива, передайте этот массив `{}`. Хотя JSX и React могут выводить массивы, объекты они не выводят; следовательно, объекты должны быть преобразованы в массив.

Разобравшись с выводом массива, можно перейти к генерированию массива элементов React. Метод `map()` хорошо подходит для этой цели, потому что он возвращает массив. Вы можете реализовать `map()` так, чтобы каждый элемент был результатом выражения `React.createElement(Link, {label: v})`, завернутого в `<div>`. В этом выражении `v` — значение элемента массива `menus` (Home, About, Services и т. д.), а `i` — его индекс (0, 1, 2, 3 и т. д.):

```
      menus.map((v, i) => {
        return React.createElement('div',
          {key: i},
          React.createElement(Link, {label: v})
        )
      }
    )
  )
}})
```

А вы заметили, что свойству `key` присваивается индекс `i`? Это необходимо, чтобы ускорить обращение к каждому элементу `<div>` в списке. Если не задать значение `key`, вы получите следующее предупреждение (по крайней мере, в React 15, 0.14 и 0.13):

```
Warning: Each child in an array or iterator should have a unique "key" prop.
Check the render method of `Menu`. See https://fb.me/react-warning-keys for
more information.
    in div (created by Menu)
    in Menu
```

И снова — респект React за хорошие сообщения об ошибках и предупреждения.

Таким образом, каждый элемент списка должен обладать уникальным значением атрибута `key`. Эти значения не обязаны быть уникальными в пределах всего приложения и других компонентов — только внутри списка. Интересно, что, начиная с версии React 15, вы не увидите атрибуты `key` в HTML (и это хорошо — не нужно загрязнять HTML). При этом ключи выводятся в React DevTools, как видно из рис. 9.2.

`<div>` имеет атрибут `key`, это важно. Он позволяет React оптимизировать рендер списков посредством преобразования их в хеши; как известно, время обращения к хешу меньше, чем к спискам или массиву. Фактически вы создаете несколько компонентов `Link` в массиве, каждый из которых получает свойство `label` со значением из массива `menus`.

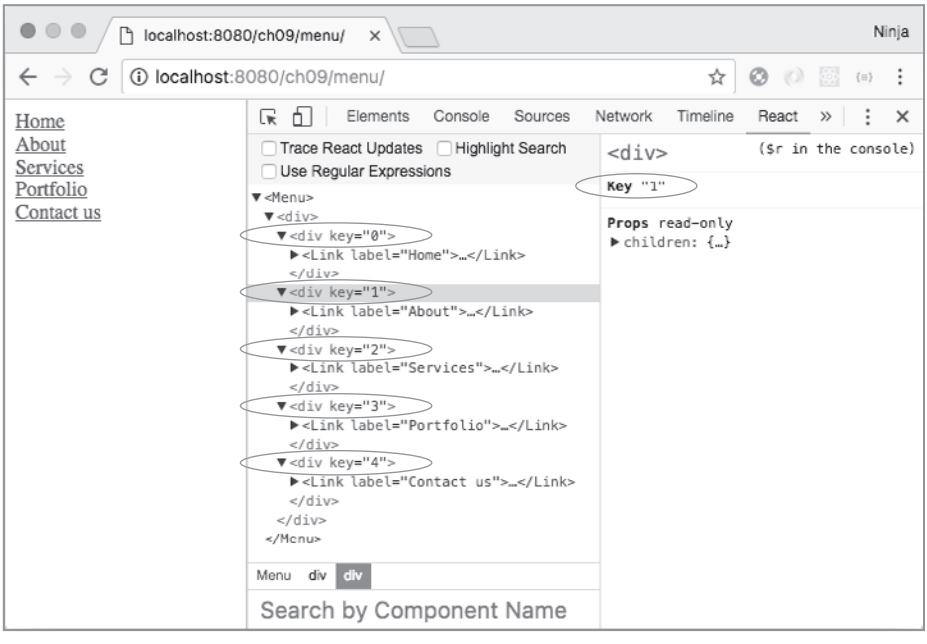


Рис. 9.2. В React DevTools выводятся ключи элементов списка

МЕТОД ARRAY.MAP()

Метод перебора (mapping) из класса `Array` часто используется в компонентах React для представления списков данных. Дело в том, что пользовательские интерфейсы часто строятся на основе данных, представленных массивом. Пользовательский интерфейс также можно рассматривать как массив, но с немного другими элементами (элементами React!).

`map()` вызывается для массива и возвращает новые элементы массива, полученные преобразованием элементов исходного массива в результате применения метода. Как минимум при работе с `map()` необходимо передать этот метод:

```
[1, 2, 3].map( value => <p>value</p>)
  ➥ // <p>1</p><p>2</p><p>3</p>
```

Кроме значения элемента (`value`) можно использовать еще два аргумента — `index` и `list`:

```
[1, 2, 3].map( (value, index, list) => {
  return <p id={index}>{list[index]}</p>
}) // <p id="0">1</p><p id="1">2</p><p id="2">3</p>
```

В листинге 9.2 приведен полный код Menu (ch09/menu/script.js); он прост и тривиален.

Листинг 9.2. Компонент Menu использует map() для рендера ссылок

```
class Menu extends React.Component {
  render() {
    let menus = ['Home',
      'About',
      'Services',
      'Portfolio',
      'Contact us']
    return React.createElement('div',
      null,
      menus.map((v, i) => {
        return React.createElement('div',
          {key: i},
          React.createElement(Link, {label: v})
        )
      })
    )
}}
```

Переходим к реализации `Link`.

9.2.2. Компонент Link

Вызов `map()` создает компонент `Link` для каждого элемента в массиве `menus`. Обратимся к коду `Link` и посмотрим, что происходит при рендере каждого компонента `Link`.

В коде рендеринга компонента `Link` пишется выражение для создания URL-адреса. Этот URL-адрес будет использоваться в атрибуте `href` тега `<a>`. Значение `this.props.label` передается `Link` из `Menu` при создании `Link`. В методе `render()` компонента `Menu` элементы компонента `Link` создаются в итерации метода `map`, используя `React.createElement(Link, {label: v})`.

Свойство `label` используется для построения описательной части (slug) URL (символы нижнего регистра, не включает пробелы):

```
class Link extends React.Component {
  render() {
    const url='/'
      + this.props.label
        .toLowerCase()
        .trim()
        .replace(' ', '-')
```

Методы `toLowerCase()`, `trim()` и `replace()` — стандартные строковые функции JavaScript. Они выполняют преобразование к нижнему регистру, усечение пробелов на концах и замену пробелов дефисами соответственно.

Выражение для URL генерирует следующие URL-адреса:

- */home* для Home
- */about* для About
- */services* для Services
- */portfolio* для Portfolio
- */contact-us* для Contact us

Теперь можно реализовать пользовательский интерфейс `Link`: возвращаемое значение `render()`. В `return` метода `render` компонента `Link` значение `this.props.label` передается как третий аргумент для `createElement()`. Оно становится частью контента тега `<a>` (текст ссылки). `Link` может отрендерить этот элемент:

```
    //...
    return React.createElement(
      'a',
      {href: url},
      this.props.label
    )
  }
}
```

Но лучше отделить каждую ссылку элементом `<br>` (перенос строки). А поскольку компонент должен вернуть только один элемент, якорный элемент (`<a>`) и перенос строки (`<br>`) нужно обернуть контейнером (`<div>`). Следовательно, команда `return` в методе `render()` компонента `Link` начинается с `div` без атрибутов:

```
//...
return React.createElement('div',
  null,
  //...
```

Каждый аргумент `createElement()` после второго (например, третий, четвертый и пятый) будет использоваться как контент (дочерние элементы). Чтобы создать элемент ссылки, передайте его во втором аргументе. А чтобы создать элемент переноса строки после каждой ссылки, передайте элемент `<br>` в четвертом аргументе:

```
    //...
    return React.createElement('div',
      null,
      React.createElement(
        'a',
        {href: url},
        this.props.label
      ),
      React.createElement('br')
    )
  }
})
```

В листинге 9.3 приведен полный код компонента `Link` (ch09/menu/script.js). Функция `url` может быть создана как метод класса или как метод за пределами компонента.

Листинг 9.3. Компонент Link

```
class Link extends React.Component {
  render() {
    const url='/'                          <-- Определяет функцию для создания фрагментов
      + this.props.label                       URL из названий команд меню
        .toLowerCase()
        .trim()
        .replace(' ', '-')
    return React.createElement('div',
      null,
      React.createElement(
        'a',
        {href: url},    <-- Передает фрагмент URL для задания атрибута href
        this.props.label
      ),
      React.createElement('br')    <-- Добавляет элемент переноса строки
    )                                  для разделения элементов меню
  }
}
```

А теперь посмотрим, как работает это меню.

9.2.3. Запуск

Чтобы увидеть страницу, показанную на рис. 9.3, откройте ее как файл в Chrome, Firefox, Safari или (возможно) Internet Explorer. Всё. Никакая компиляция для этого проекта не нужна.

ИСПОЛЬЗОВАНИЕ ЛОКАЛЬНОГО ВЕБ-СЕРВЕРА

Когда вы откроете файл со страницей, в адресной строке будет отображаться протокол file://... — не идеально, но сойдет для нашего проекта. Для реальной разработки вам понадобится веб-сервер; с веб-сервером будет отображаться протокол http://... или https://..., как показано на рис. 9.3.

Да, даже для таких простых веб-страниц я предпочитаю использовать локальный веб-сервер. С ним выполнение кода ближе к тому, что будет происходить в условиях реальной эксплуатации. Кроме того, вы можете использовать AJAX/XHR, что невозможно при открытии файла HTML в браузере.

Для запуска локального веб-сервера проще всего воспользоваться node-static (www.npmjs.com/package/node-static) или другим аналогичным инструментом Node.js, например http-server (www.npmjs.com/package/http-server). Это относится даже к Windows, хотя я перестал использовать эту ОС много лет назад. Если вы решительно против использования Node.js, другие возможные варианты — IIS,

Apache HTTP Server, NGINX, MAMP, LAMP и другие разновидности веб-серверов. Не стоит и говорить, что я настоятельно рекомендую использовать инструменты Node.js из-за их минимализма и эффективности.

Чтобы установить node-static, воспользуйтесь npm:

```
$ npm install -g node-static@0.7.6
```

После установки выполните следующую команду из корневой папки вашего проекта (или родительской папки), чтобы файл стал доступен по адресу http://localhost:8080. Этот адрес не является внешней ссылкой — выполните следующую команду перед щелчком на ссылке:

```
$ static
```

Если выполнить `static` из react-quickly/ch09/menu, URL будет иметь вид http://localhost:8080. С другой стороны, если выполнить `static` из react-quickly, то URL должен иметь вид http://localhost:8080/ch09/menu.

Чтобы остановить сервер на macOS или UNIX/Linux (системы POSIX), нажмите Ctrl+C. А насчет Windows — я не в курсе!

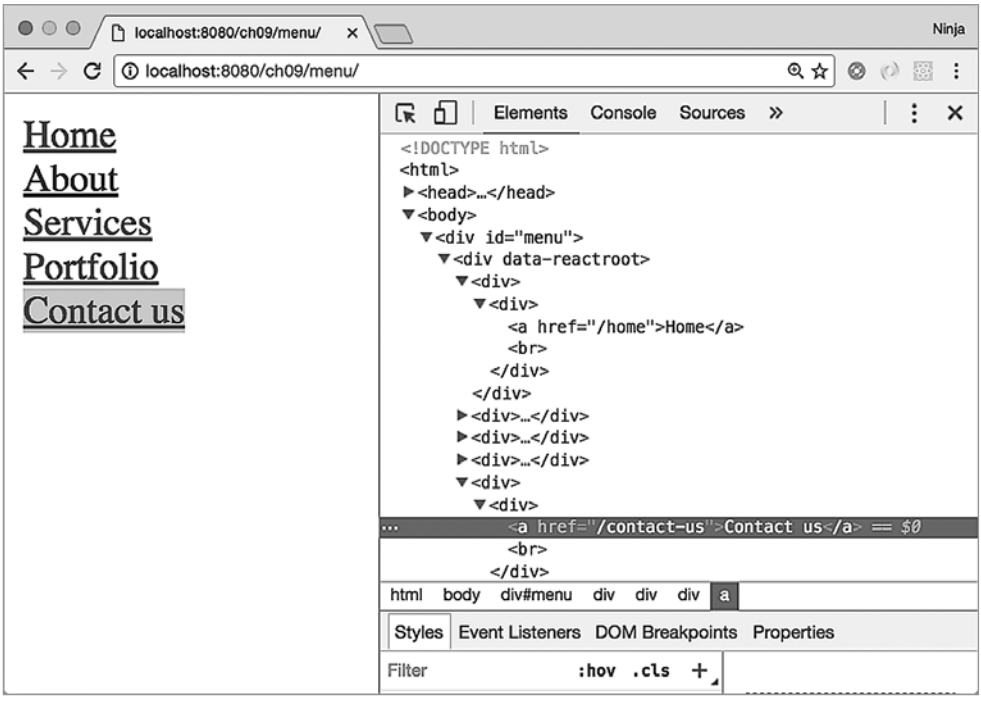


Рис. 9.3. Меню React с прорисовкой вложенных компонентов

Ничего сверхъестественного, но на странице должны отображаться пять ссылок (или больше, если вы добавите элементы в массив `menus`), как было показано на рис. 9.1. Это намного лучше, чем копировать и вставлять пять элементов `<a>`, а потом изменять надписи и URL в пяти разных местах. А с JSX проект станет еще лучше!

9.3. Построение меню в JSX

Этот проект будет более обширным: в нем появляется папка node_modules, файл package.json и JSX:

```
/menu-jsx
  /node_modules  ◄——  Зависимость Babel для транспиляции JSX в JS
  index.html
  package.json
  react-dom.js
  react.js
  script.js
  script.jsx  ◄——  Основной сценарий JSX
```

Как видите, в проекте появилась папка node_modules для пакетов разработки — например, зависимости Babel, используемой для транспиляции JSX в JS.

> **ПРИМЕЧАНИЕ** Хотя react и react-dom можно установить как npm-модули, вместо того чтобы хранить их в виде файлов, это создаст дополнительные сложности при развертывании. В данный момент для развертывания вашего приложения достаточно скопировать папку проекта без `node_modules`. Если же вы установите React и ReactDOM при помощи npm, вам также придется включить эту папку, воспользоваться сборщиком (bundler) или скопировать файлы JS из дистрибутива в корневую папку (где они уже находятся). Поэтому в своем примере мы будем использовать файлы в корневой папке. Сборщики рассматриваются в части 2 этой книги, но пока обойдемся без лишних сложностей.

Создайте новую папку:

```
$ mkdir menu-jsx
$ cd menu-jsx
```

Затем создайте в ней файл package.json командой `npm init -y`. Включите код из листинга 9.4 в package.json, чтобы установить и настроить Babel (ch09/menu-jsx/package.json).

Листинг 9.4. Файл package.json для Menu в JSX

```
{
  "name": "menu-jsx",
  "version": "1.0.0",
  "description": "",
  "main": "script.js",
```

```
  "scripts": {
    "build": "./node_modules/.bin/babel script.jsx -o script.js -w"
  },
  "author": "Azat Mardan",
  "license": "MIT",
  "babel": {
    "presets": ["react"]
  },
  "devDependencies": {
    "babel-cli": "6.9.0",
    "babel-preset-react": "6.5.0"
  }
}
```

Определяет сценарий сборки с флагом отслеживания изменений

Настраивает Babel для транспиляции JSX React

Включает Babel CLI и конфигурацию React/JSX

Установите пакеты для разработки командой `npm i` или `npm install`. На этом подготовка должна быть завершена.

Теперь рассмотрим файл script.jsx. На верхнем уровне он состоит из следующих частей:

```
class Menu extends React.Component {
  render() {
    //...
  }
}

class Link extends React.Component {
  render() {
    //...
  }
}

ReactDOM.render(<Menu />, document.getElementById('menu'))
```

Выглядит знакомо, не так ли? Та же структура, что и в `Menu` без JSX. Главное изменение в этом высокоуровневом листинге — замена `createElement()` для компонента `Menu` в `ReactDOM.render()` следующей строкой:

```
ReactDOM.render(<Menu />, document.getElementById('menu'))
```

А теперь займемся переработкой компонентов.

9.3.1. Переработка компонента Menu

Начало `Menu` остается неизменным:

```
class Menu extends React.Component {
  render() {
    let menus = ['Home',
```

```
      'About',
      'Services',
      'Portfolio',
      'Contact us']
    return //...
  }
}
```

В переработанном коде компонента `Menu` значение `v` должно выводиться как значение атрибута `label` (то есть `label={v}`). Другими словами, значение `v` присваивается как свойство для `label`. Таким образом, строка создания элемента `Link`

```
React.createElement(Link, {label: v})
```

превращается в следующий JSX-код:

```
<Link label={v}/>
```

Свойство `label` второго аргумента (`{label: v}`) становится атрибутом `label={v}`. Значение `v` атрибута объявляется в `{}`, чтобы оно было динамическим (в отличие от жестко фиксированного значения).

> **ПРИМЕЧАНИЕ** При использовании фигурных скобок для присваивания значений свойств двойные кавычки (`""`) не нужны.

React также необходим атрибут `key={i}` для более эффективного обращения к элементам списка. По этой причине компонент `Menu` после реструктуризации превращается в следующий код JSX (ch09/menu-jsx/script.jsx).

Листинг 9.5. Меню с JSX

```
class Menu extends React.Component {
  render() {
    let menus = ['Home',
      'About',
      'Services',
      'Portfolio',
      'Contact us']
    return <div>
      {menus.map((v, i) => {
        return <div key={i}><Link label={v}/></div>
      })}
    </div>
}}
```

Как вы думаете, этот код лучше читается? На мой взгляд, да!

В методе `render()` компонента `Menu`, если вы предпочитаете начинать элемент `<div>` в новой строке, заключите его в круглые скобки `()`. Например, следующий код

идентичен листингу 9.5, но `<div>` начинается в новой строке; этот вариант лучше смотрится:

```
  //...
    return (
      <div>
        {menus.map((v, i) => {
          return <div key={i}><Link label={v}/></div>
        })}
      </div>
    )
}})
```

9.3.2. Переработка компонента Link

Теги `<a>` и `<br>` в компоненте `Link` также преобразуются из исходного вида:

```
//...
  return React.createElement('div',
    null,
    React.createElement(
      'a',
      {href: url},
      this.props.label),
    React.createElement('br')
    )
//...
```

в следующий код JSX:

```
//...
  return <div>
    <a href={url}>
    {this.props.label}
    </a>
    <br/>
  </div>
//...
```

В листинге 9.6 приведена полная JSX-версия компонента `Link` (ch09/menu-jsx/script.jsx).

Листинг 9.6. JSX-версия компонента Link

```
class Link extends React.Component {
  render() {
    const url='/'
      + this.props.label
        .toLowerCase()
        .trim()
        .replace(' ', '-')
```

```
    return <div>
      <a href={url}>
      {this.props.label}
      </a>
      <br/>
    </div>
  }
}
```

Вот и всё! Теперь проект JSX нужно запустить.

9.3.3. Запуск проекта JSX

Откройте терминал, iTerm или приложение окна командной строки. В папке проекта (ch09/menu-jsx или то имя, которое было присвоено при загрузке исходного кода) установите зависимости командой `npm i` (сокращение для `npm install`) после записей в package.json.

Затем запустите сценарий сборки npm командой `npm run build`. Сценарий npm выполнит команду Babel с флагом отслеживания изменений (`-w`), который будет держать Webpack запущенным, так что он сможет отслеживать любые изменения в файлах и перекомпилировать код из JSX в JS при наличии изменений в исходном коде JSX.

Не стоит и говорить, что режим отслеживания изменений экономит время, потому что он избавляет от необходимости перекомпиляции при каждом изменении в исходном коде. Оперативная замена модулей еще полезнее для разработки (настолько, что она сама по себе может стать веской причиной для использования React); мы рассмотрим ее в главе 12.

Реальная команда в сценарии сборки выглядит так (но кому захочется ее вводить — слишком длинная!):

```
./node_modules/.bin/babel script.jsx -o script.js -w
```

Если вы захотите освежить в памяти Babel CLI, обращайтесь к главе 3. В ней вы найдете все подробности.

На моем компьютере Babel CLI выдает следующее сообщение (на вашем компьютере путь будет другим):

```
> menu-jsx@1.0.0 build /Users/azat/Documents/Code/react-quickly/ch09/menu-jsx
> babel script.jsx -o script.js -w
```

Всё готово. Со сгенерированным сценарием script.js вы можете воспользоваться командой `static` (`node-static` в npm: `npm i -g node-static`) для предоставления доступа к файлам через HTTP на хосте `localhost`. Приложение должно выглядеть и работать точно так же, как его обычные собратья JavaScript (рис. 9.4).

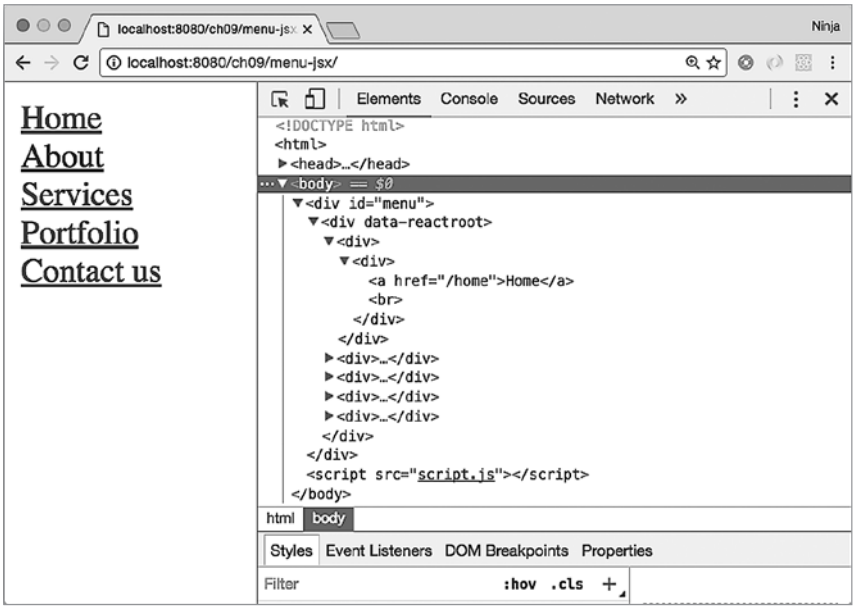


Рис. 9.4. Меню, созданное в JSX

9.4. Домашнее задание

Для получения дополнительных баллов сделайте следующее:

- Загрузите меню из menus.json с использованием Fetch API. О том, как загружать данные, рассказано в главе 5.
- Создайте сценарий npm, который будет получать react.js из npm-пакета react, установленного в node_modules, и копировать его в папку проекта для использования файлом index.html. Это избавит вас от необходимости вручную загружать react.js для будущих версий; вместо этого вы сможете использовать команду `npm i react`, а потом запустить сценарий.

Отправьте свой код *в новую папку в* ch09 как pull-запрос в репозиторий GitHub этой книги: https://github.com/azat-co/react-quickly.

9.5. Итоги

- `key` — ваш друг. Устанавливайте этот атрибут при генерировании списков.
- `map()` — элегантный способ создания нового массива на основе исходного массива. Он получает аргументы `value`, `index` и `list`.
- Для работы JSX как минимум необходимы конфигурации Babel CLI и React.

10 Компонент Tooltip

Посмотрите вступительный видеоролик к этой главе, отсканировав QR-код или перейдя по ссылке http://reactquickly.co/videos/ch10.

ЭТА ГЛАВА ОХВАТЫВАЕТ СЛЕДУЮЩИЕ ТЕМЫ:

- Структура проекта и инициализация.
- Компонент Tooltip.

При работе с веб-сайтами, содержащими большой объем текста (например, википедией), для пользователя очень удобно получать дополнительную информацию без потери позиции и контекста. Например, дополнительная информация может выводиться на панели при наведении указателя мыши (рис. 10.1). Эта панель называется *экранной подсказкой* (tooltip).

Вся суть React — построение и совершенствование пользовательских интерфейсов, поэтому реализация экранной подсказки будет вполне уместной. Давайте построим компонент для вывода содержательного текста по событию наведения указателя мыши.

Существует несколько готовых реализаций экранных подсказок, включая `react-tooltip` (www.npmjs.com/package/react-tooltip), но мы создадим такую реализацию самостоятельно для изучения React. Построение экранной подсказки с нуля — *очень* хорошее упражнение. Возможно, вы будете использовать этот пример в своей повседневной работе, включив его в свое приложение, а то и доработаете его до нового компонента React с открытым кодом!

Вся суть создания компонента `Tooltip` — получить произвольный текст, скрыть его средствами CSS и снова сделать видимым по событию наведения. В этом проекте будут использоваться условия `if/else`, JSX и другие элементы программирования. В части CSS вам понадобятся классы Twitter Bootstrap и специальная тема Twitter Bootstrap, которая позволит придать подсказке приличный вид за короткое время.

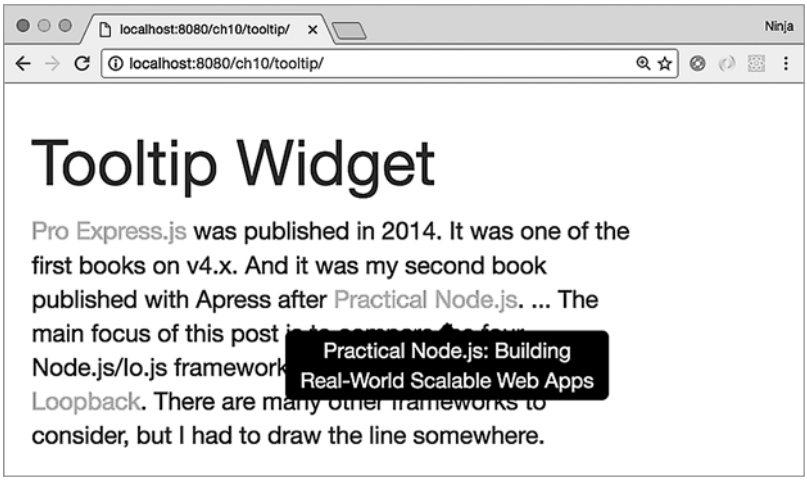


Рис. 10.1. Подсказка появляется при наведении указателя мыши на помеченный текст

ПРИМЕЧАНИЕ Чтобы повторить приведенное описание работы над проектом, необходимо загрузить неминифицированную версию React и установить node.js и npm для компиляции JSX. В этом примере также используется тема Flatly из Bootswatch (https://bootswatch.com/flatly). Эта тема зависит от Twitter Bootstrap. Установка всех программ описана в приложении А.

ПРИМЕЧАНИЕ Исходный код примеров этой главы доступен по адресам www.manning.com/books/react-quickly и https://github.com/azat-co/react-quickly/tree/master/ch10 (в папке ch10 репозитория GitHub https://github.com/azat-co/react-quickly). Также некоторые демонстрационные примеры доступны по адресу http://reactquickly.co/demos.

10.1. Структура проекта и инициализация

Проект компонента `Tooltip` имеет следующую структуру:

```
/tooltip
  /node_modules  <-- Зависимость Babel для транспиляции JSX в JS
  bootstrap.css
  index.html
  package.json
  react-dom.js
  react.js
  script.js
  script.jsx  <-- Основной сценарий JSX
```

Как и в главе 9, в проекте имеется папка node_modules для зависимостей разработки, например зависимости Babel, используемой для транспиляции JSX в JS. Проект

имеет плоскую структуру — стили и сценарии хранятся в одной папке. Это было сделано для простоты. Конечно, в реальном проекте стили и сценарии будут находиться в разных папках.

Важнейшие части package.json — сценарий npm для сборки, конфигурация Babel, зависимости и другие метаданные.

Листинг 10.1. Файл package.json проекта Tooltip

```
{
  "name": "tooltip",
  "version": "1.0.0",
  "description": "",
  "main": "script.js",
  "scripts": {
    "build": "./node_modules/.bin/babel script.jsx -o script.js -w"
  },
  "author": "Azat Mardan",
  "license": "MIT",
  "babel": {
    "presets": ["react"]
  },
  "devDependencies": {
    "babel-cli": "6.9.0",
    "babel-preset-react": "6.5.0"
  }
}
```

После создания файла package.json не забудьте выполнить команду `npm i` или `npm install`.

Далее можно переходить к HTML. Создайте файл index.html, приведенный в листинге 10.2 (ch10/tooltip/index.html).

Листинг 10.2. Файл index.html проекта Tooltip

```
<!DOCTYPE html>
<html>

  <head>
    <script src="react.js"></script>
    <script src="react-dom.js"></script>
    <link href="bootstrap.css"  <-- Применяет стили
      rel="stylesheet"
      type="text/css"/>
    </head>

    <body class="container">
      <h1>Tooltip Widget</h1>
      <div id="tooltip"></div>  <-- Определяет элемент рендеринга для React и Tooltip
      <script src="script.js" type="text/javascript"></script>
    </body>

</html>
```

В секции `<head>` включаются файлы React, React DOM и стили Twitter Bootstrap. Секция `<body>` минимальна: она содержит элемент `<div>` с идентификатором `tooltip` и файл script.js приложения.

На следующем шаге будет создан файл script.jsx. Все верно — здесь нет опечатки. Исходный код хранится в файле script.jsx, а в разметку включается файл script.js. Дело в том, что вы будете использовать инструменты командной строки Babel.

10.2. Компонент Tooltip

Обратимся к файлу script.jsx (ch10/tooltip/script.jsx). Он содержит почти весь код компонента и текст подсказки, которая должна появиться на экране. Текст подсказки представляет собой свойство, которое задается при создании `Tooltip` в `ReactDOM.render()`.

Листинг 10.3. Компонент Tooltip и текст

```
class Tooltip extends React.Component {
  constructor(props) {
    ...
  }
  toggle() {    <-- Объявляет метод для отображения и сокрытия текста
    ...
  }
  render() {    <-- Объявляет обязательный метод render()
    ...
  }
}

ReactDOM.render(<div>
  <Tooltip text="The book you're reading now">React Quickly</Tooltip>
    was published in 2017. It's awesome!
  </div>,
  document.getElementById('tooltip'))
```

Передает текст подсказки в свойстве. Содержимое — помеченный текст, над которым пользователь задерживает указатель мыши

Реализуем `Tooltip` и объявим компонент с исходным состоянием `opacity: false`. Это состояние управляет сокрытием или отображением текста. (В главе 4 состояния рассматриваются более подробно.) Метод `constructor()` выглядит так:

```
class Tooltip extends React.Component {
  constructor(props) {
    super(props)
    this.state = {opacity: false}
    this.toggle = this.toggle.bind(this)
  }
  ...
}
```

В исходном состоянии текст подсказки скрывается. Функция `toggle()` изменяет состояние видимости подсказки на противоположную, то есть управляет отображением текста подсказки. Перейдем к реализации `toggle()`.

10.2.1. Функция toggle()

Теперь мы определим функцию `toggle()`, которая изменяет видимость подсказки, меняя состояние `opacity` на противоположное (`true` меняется на `false`, а `false` на `true`):

```
toggle() {
    const tooltipNode = ReactDOM.findDOMNode(this)
    this.setState({
      opacity: !this.state.opacity,
      ...
    })
}
```

Для изменения `opacity` используется метод `this.setState()`, о котором вы узнали в главе 4.

Хитрость с текстом подсказки состоит в том, что он должен выводиться рядом с элементом, на который наведен указатель мыши. Для этого нужно получить позицию компонента с использованием `tooltipNode`. Для позиционирования текста подсказки используются `offsetTop` и `offsetLeft` — свойства узла DOM из стандарта HTML (https://developer.mozilla.org/en-US/docs/Web/API/Node), не из React:

```
    top: tooltipNode.offsetTop,
    left: tooltipNode.offsetLeft
  })
},
```

В листинге 10.4 приведен полный код `toggle()` (ch10/tooltip/script.jsx).

Листинг 10.4. Функция toggle()

```
toggle() {
  const tooltipNode = ReactDOM.findDOMNode(this)
    this.setState({
      opacity: !this.state.opacity,
      top: tooltipNode.offsetTop,
      left: tooltipNode.offsetLeft
    })
}
```

А вот она же с использованием деструктуризации ES:

```
toggle() {
  const {offsetTop: top, offsetLeft: left} = ReactDOM.findDOMNode(this)

  this.setState({
```

```
    opacity: !this.state.opacity,
      top,
      left
  })
}
```

Из кода видно, что он изменяет состояние и позицию. Нужно ли заново рендерить представление? Нет, потому что React обновит представление за вас. `setState()` вызовет повторный ренедер автоматически. Это может привести к изменениям в DOM, а может и не привести — это зависит от того, было ли состояние использовано в `render()`. Реализацией этой функции мы займемся сейчас.

10.2.2. Функция render()

Функция `render()` содержит CSS-объект `style` для текста подсказки, а также стили Twitter Bootstrap. Сначала необходимо определить объект `style`. Стили CSS `opacity` и `z-index` задаются в зависимости от значения `this.state.opacity`. Значение `z-index` понадобится для того, чтобы подсказка размещалась над всеми остальными элементами, поэтому его следует задать достаточно высоким, например: `1000` для видимого текста и `-1000` для скрытого:

```
zIndex: (this.state.opacity) ? 1000 : -1000,
```

Для `z-index` необходимо использовать `zIndex` (обратите внимание на «верблюжий регистр»). На рис. 10.2 показано, как применяются стили в событии наведения указателя мыши (`opacity` содержит `true`).

> **СОВЕТ** Не забудьте использовать «верблюжий регистр» с React вместо синтаксиса с дефисом. Свойство CSS `z-index` превращается в свойство стиля React `zIndex`; `background-color` превращается в `backgroundColor`; `font-family` превращается в `fontFamily` и т. д. При использовании действительных имен JavaScript React сможет быстрее обновить реальную модель DOM из виртуальной.

Состояние непрозрачности `this.state.opacity` представляет собой логическое значение `true` или `false`, но непрозрачность CSS является двоичным значением 0 или 1. Если состояние непрозрачности равно `false`, непрозрачность CSS равна 0; а если состояние непрозрачности равно `true`, непрозрачность CSS равна 1. Преобразование выполняется при помощи бинарного оператора (`+`):

```
opacity: +this.state.opacity,
```

Что касается позиции подсказки, текст должен располагаться неподалеку от текста, на который был наведен указатель мыши. Для этого мы добавим 20 пикселов к верхнему краю (расстояние от верхней стороны окна до элемента) и вычтем 30 пикселов из левого края (расстояние от левой стороны окна до элемента). Значения были выбраны на глаз; вы можете изменить логику так, как считаете нужным:

```
render() {
  const style = {
    zIndex: (this.state.opacity) ? 1000 : -1000,
    opacity: +this.state.opacity,
    top: (this.state.top || 0) + 20,
    left: (this.state.left || 0) -30
  }
```

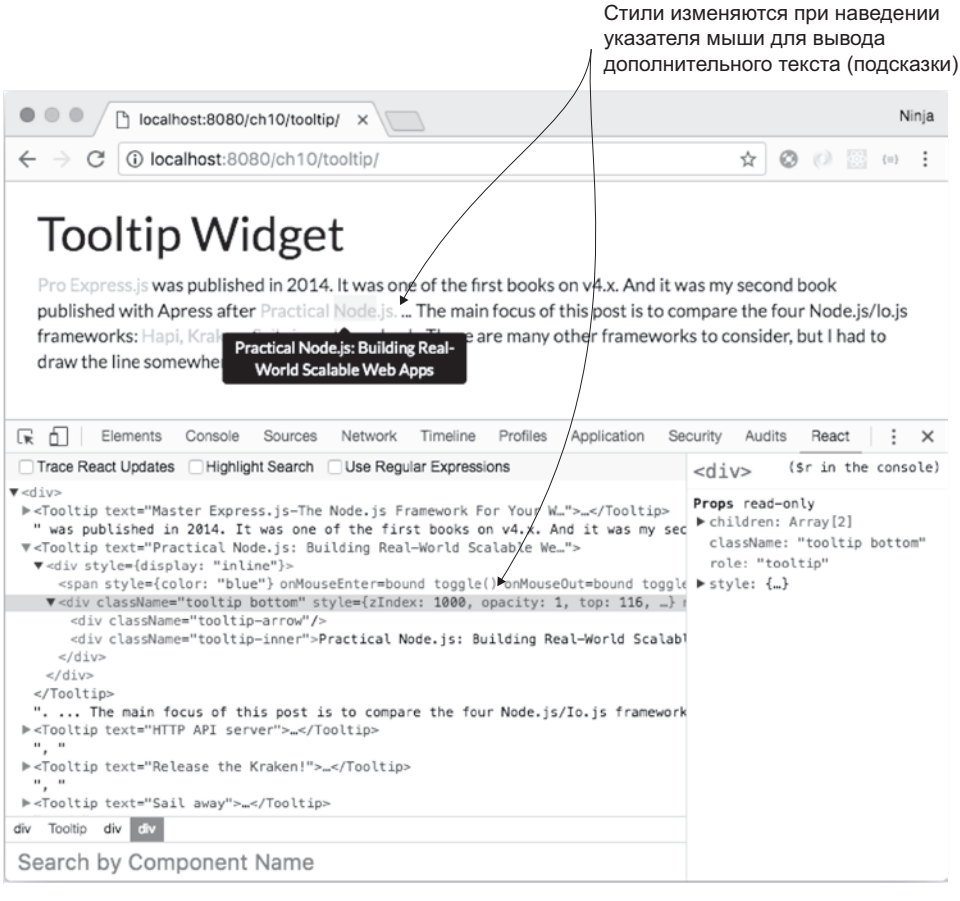


Рис. 10.2. Текст подсказки выводится при наведении указателя мыши, для чего opacity присваивается значение 1, а zIndex — значение 1000

Затем идет `return`. Компонент рендерит как текст, на который наводится указатель мыши, так и текст справки. Я использую классы Twitter Bootstrap в сочетании с моим объектом `style`, чтобы скрыть текст подсказки и отобразить его позднее.

Текст, на который наводится указатель для вывода подсказки, окрашен в синий цвет, чтобы он визуально отличался от другого текста. С ним связаны два события мыши — для входа и выхода указателя:

```
return (
  <div style={{display: 'inline'}}>
    <span style={{color: 'blue'}}
      onMouseEnter={this.toggle}
      onMouseOut={this.toggle}>
      {this.props.children}
    </span>
```

Выводит все, что внутренняя разметка HTML передаст Tooltip позднее

Далее идет код текста подсказки. Он почти статичен, за исключением `{style}`. React изменит состояние, а это инициирует изменение в пользовательском интерфейсе:

Использует класс для направленной стрелки

Применяет объект стиля к атрибуту style

```
<div className="tooltip bottom" style={style} role="tooltip">
          <div className="tooltip-arrow"></div>
          <div className="tooltip-inner">
            {this.props.text}
          </div>
        </div>
      </div>
    )
  }
}
```

Выводит текст подсказки из свойства text {this.props.text}

В листинге 10.5 приведен полный код функции `render()` компонента `Tooltip`.

Листинг 10.5. Полный код функции render() для Tooltip

```
render() {
    const style = {
      zIndex: (this.state.opacity) ? 1000 : -1000,
      opacity: +this.state.opacity,
      top: (this.state.top || 0) + 20,
      left: (this.state.left || 0) - 30
    }
    return (
      <div style={{display: 'inline'}}>
        <span style={{color: 'blue'}}
          onMouseEnter={this.toggle}
          onMouseOut={this.toggle}>
          {this.props.children}
        </span>
        <div className="tooltip bottom"
          style={style}
          role="tooltip">
          <div className="tooltip-arrow"></div>
          <div className="tooltip-inner">
            {this.props.text}
          </div>
        </div>
      </div>
    )
  }
```

Инициирует вывод подсказки при входе указателя мыши

Выводит текст, переданный как содержимое Tooltip

Применяет стили с opacity, zIndex и правильной позицией на основании позиции узла DOM

Выводит текст подсказки с использованием классов Twitter Bootstrap

Вот и всё. Компонент `Tooltip` готов!

10.3. Запуск

Попробуйте использовать компонент в своих проектах, скомпилировав JSX из npm:

```
$ npm run build
```

Компонент `Tooltip` хорошо выглядит благодаря стилям Twitter Bootstrap. Возможно, он не настолько гибок, как некоторые существующие модули, но вы самостоятельно построили его с нуля. Понимаете, к чему я? С помощью классов Twitter Bootstrap и React вы можете создать хорошую подсказку (рис. 10.3) почти моментально. Более того, компонент адаптируется к разным размерам экранов благодаря динамическому позиционированию!

Рис. 10.3. Когда пользователь наводит указатель мыши на синий текст, появляется черный контейнер с текстом и стрелкой, который содержит дополнительную информацию

10.4. Домашнее задание

Для получения дополнительных баллов сделайте следующее:

- Реализуйте модификацию, которая работает по щелчку кнопки мыши, то есть выводит подсказку, когда вы щелкаете на помеченном тексте, и скрывает ее при повторном щелчке.

- Доработайте компонент `Tooltip` так, чтобы он получал свойство, определяющее поведение (по наведении указателя мыши или по щелчку).
- Доработайте компонент `Tooltip` так, чтобы он получал свойство, позиционирующее текст подсказки над текстом (вместо позиции по умолчанию под текстом. Для этого измените класс `TB` и свойства `top` и `left`).

Отправьте свой код в новую папку в `ch10` как pull-запрос в репозиторий GitHub этой книги: https://github.com/azat-co/react-quickly.

10.5. Итоги

- Стилевые свойства React записываются в «верблюжьем регистре» в отличие от стилевых свойств CSS.
- `this.props.children` определяет содержимое компонента.
- В ручном повторном рендеринге нет необходимости, потому что React автоматически выполняет ререндер после `setState()`.

11 Компонент Timer

Посмотрите вступительный видеоролик к этой главе, отсканировав QR-код или перейдя по ссылке http://reactquickly.co/videos/ch11.

ЭТА ГЛАВА ОХВАТЫВАЕТ СЛЕДУЮЩИЕ ТЕМЫ:

- Структура проекта и инициализация.
- Архитектура приложения.

Исследования показали, что медитация очень полезна для здоровья (успокоение) и эффективности (концентрация[1]). А кому не хочется обрести здоровье и работать более производительно, особенно с минимальными затратами?

Гуру рекомендуют начинать с пятиминутной медитации, потом увеличить ее продолжительность до 10 минут, а затем довести ее до 15 минут через несколько недель. Стремиться следует к 30–60 минутам медитации в день, но некоторые люди замечают улучшения уже при 10 минутах. Я могу подтвердить: после десятиминутных ежедневных медитаций в течение трех лет я стал более внимательным; кроме того, это помогло мне в других областях.

Но как узнать, когда будет достигнута ваша ежедневная цель по продолжительности медитации? Вам нужен таймер! В этой главе вы проверите свои навыки React и HTML5 и создадите веб-таймер (рис. 11.1). Чтобы упростить тестирование примера, этот таймер может запускаться только на 5, 10 и 15 секунд.

---

[1] См. «Research on Meditation», Википедия, https://en.wikipedia.org/wiki/Research_on_meditation; «Meditation: In Depth», Национальные институты здравоохранения, http://mng.bz/01om; «Harvard Neuroscientist: Meditation Not Only Reduces Stress, Here's How It Changes Your Brain», The Washington Post, 26 мая 2015 г., http://mng.bz/1ljZ; «Benefits of Meditation», Yoga Journal, http://mng.bz/7Hp7.

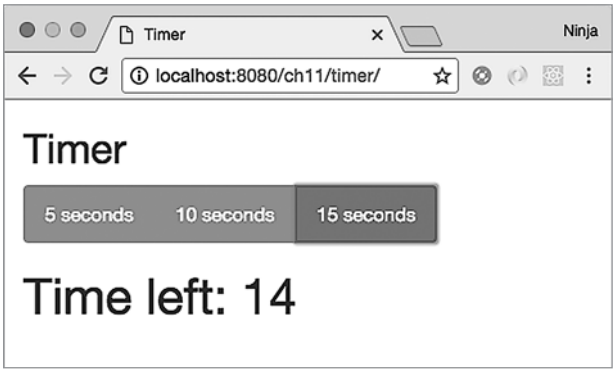


Рис. 11.1. Таймер в действии: осталось всего 14 секунд. На выбранной кнопке 15 Seconds щелчок был сделан всего секунду назад

Идея заключается в том, чтобы создать три элемента управления для запуска таймера с обратным отчетом (от *n* до 0). Представьте типичный кухонный таймер, но вместо минут он отсчитывает секунды. Вы щелкаете по кнопке — таймер запускается. Щелкаете снова (или щелкаете по другой кнопке) — таймер снова запускается.

ПРИМЕЧАНИЕ Чтобы повторить приведенное описание работы над проектом, необходимо загрузить неминифицированную версию React и установить node.js и npm для компиляции JSX. В этом примере также используется тема Flatly из Bootswatch (https://bootswatch.com/flatly). Эта тема зависит от Twitter Bootstrap. Установка всех программ описана в приложении A.

ПРИМЕЧАНИЕ Исходный код примеров этой главы доступен по адресам www.manning.com/books/react-quickly и https://github.com/azat-co/react-quickly/tree/master/ch11 (в папке ch11 репозитория GitHub https://github.com/azat-co/react-quickly). Также некоторые демонстрационные примеры доступны по адресу http://reactquickly.co/demos.

11.1. Структура проекта и инициализация

Проект компонента `Timer` имеет почти такую же структуру, как `Tooltip` и `Menu`:

```
/timer
  /node_modules   <-- Зависимость Babel для транспиляции JSX в JS
  bootstrap.css
  flute_c_long_01.wav   <-- Аудиофайл для выдачи сигнала об истечении времени
  index.html
  package.json
  react-dom.js
  react.js
  timer.js
  timer.jsx   <-- Основной сценарий JSX
```

Как и прежде, в проекте имеется папка node_modules для зависимостей разработки, например зависимости Babel, используемой для транспиляции JSX в JS. Проект имеет плоскую структуру — стили и сценарии хранятся в одной папке. Это было сделано для простоты. Конечно, в реальном проекте стили и сценарии будут находиться в разных папках.

Важнейшие части package.json — сценарий npm для сборки, конфигурация Babel, зависимости и другие метаданные.

Листинг 11.1. Проект Timer в package.json

```
{
  "name": "timer",
  "version": "1.0.0",
  "description": "",
  "main": "script.js",
  "scripts": {
    "build": "./node_modules/.bin/babel timer.jsx -o timer.js -w"
  },
  "author": "Azat Mardan",
  "license": "MIT",
  "babel": {
    "presets": ["react"]
  },
  "devDependencies": {
    "babel-cli": "6.9.0",
    "babel-preset-react": "6.5.0"
  }
}
```

Создает сценарий npm для транспиляции JSX в JS

После того как вы создадите файл package.json () (введете вручную или скопируете из архива), не забудьте выполнить команду `npm i` или `npm install`.

Разметка HTML этого проекта чрезвычайно проста (ch11/timer/index.html). Она включает файлы react.js и react-dom.js, которые ради простоты размещаются в одной папке с файлом HTML.

Листинг 11.2. Файл index.html из проекта Timer

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Timer</title>
    <script src="react.js" type="text/javascript"></script>
    <script src="react-dom.js" type="text/javascript"></script>
    <link href="bootstrap.css" rel="stylesheet" type="text/css"/>
  </head>
  <body class="container-fluid">
    <div id="timer-app"/>
  </body>
  <script src="timer.js" type="text/javascript"></script>
</html>
```

Этот файл только включает библиотеку и добавляет ссылку на файл timer.js, который будет создан из timer.jsx. Для этого вам понадобится Babel CLI (см. главу 3).

11.2. Архитектура приложения

Файл timer.jsx будет содержать три компонента:

- `TimerWrapper` — основной компонент, который выполняет большую часть работы и рендерит другие компоненты.
- `Timer` — компонент для вывода числа оставшихся секунд.
- `Button` — компонент для рендера трех кнопок и запуска (сброса) таймера.

На рис. 11.2 показано, как эти компоненты выглядят на странице. Компоненты `Timer` и `Button` видны на странице; `TimerWrapper` содержит все три кнопки и `Timer`. `TimerWrapper` — контейнерный компонент, тогда как два других компонента — презентационные.

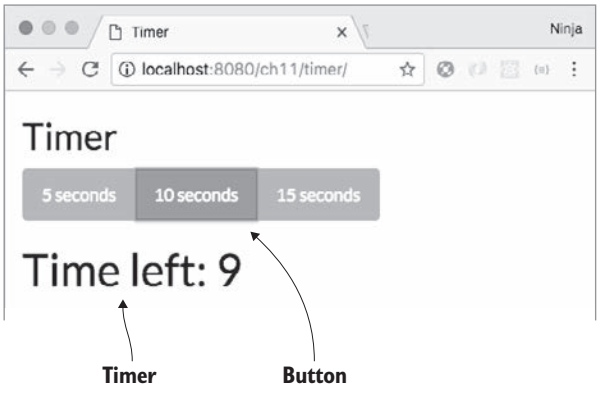


Рис. 11.2. Компоненты Timer и Button

Мы разобьем приложение на три части, потому что в области программирования многие аспекты изменяются с каждой новой версией. Если уровень представления (`Button` и `Timer`) отделяется от логики (`TimerWrapper`), вашему приложению будет проще адаптироваться к изменениям. Более того, такие элементы, как кнопки, можно будет повторно использовать в других приложениях. В общем, разделение уровня представления и бизнес-логики считается рекомендуемой практикой при работе с React.

Компонент `TimerWrapper` обеспечивает взаимодействие между `Timer` и компонентами `Button`. Схема взаимодействия между тремя компонентами и пользователем изображена на рис. 11.3.

1. `TimerWrapper` рендерит `Timer` и `Button`, передавая состояния `TimeWrapper` в свойствах.
2. Пользователь нажимает кнопку, что приводит к инициированию события кнопкой.
3. Событие кнопки вызывает функцию из `TimerWrapper` с продолжительностью интервала в секундах.
4. `TimerWrapper` задает интервал и обновляет `Timer`.
5. Обновления продолжаются до тех пор, пока не останется 0 секунд.

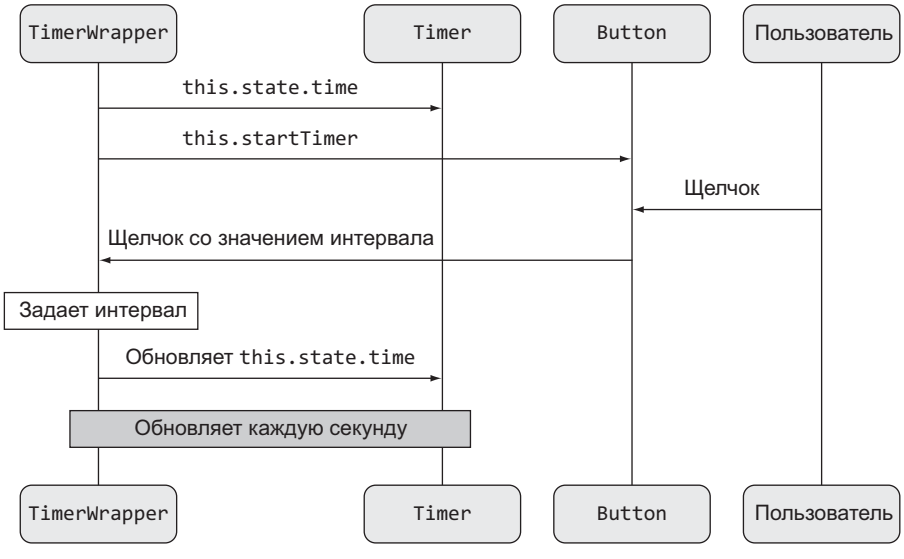


Рис. 11.3. Логика выполнения приложения Timer

Для простоты все три компонента будут храниться в одном файле timer.jsx.

Листинг 11.3. Структура timer.jsx

```
class TimerWrapper extends React.Component {
  constructor(props) {   <-- Задает исходные состояния
    // ...
  }
  startTimer(timeLeft) {   <-- Запускает новый таймер (сброс)
    // ...
  }
  render() {
    // ...
  }
}
```

```
class Timer extends React.Component {
  render() {
    // ...
  }
}

class Button extends React.Component {
  startTimer(event) {
    // ...
  }
  render() {
    // ...
  }
}

ReactDOM.render(
  <TimerWrapper/>,
  document.getElementById('timer-app')
)
```

Запускает новый таймер (сброс) из события щелчка. Вызывает startTimer из TimerWrapper

Выполняет рендер TimerWrapper

Начнем с нижней части файла timer.jsx и выполним рендеринг главного компонента (`TimerWrapper`) в `<div>` с идентификатором `timer-app`:

```
ReactDOM.render(
  <TimerWrapper/>,
  document.getElementById('timer-app')
)
```

`ReactDOM.render()` будет последним вызовом в файле. Он использует `TimerWrapper`, поэтому этот компонент будет определен следующим.

11.3. Компонент TimerWrapper

Всё самое интересное происходит в `TimerWrapper`! Высокоуровневая структура компонента выглядит так:

```
class TimerWrapper extends React.Component {
  constructor(props) {
    // ...
  }
  startTimer(timeLeft) {
    // ...
  }
  render() {
    // ...
  }
}
```

Для начала нужно иметь возможность сохранить оставшееся время (`timeLeft`) и сбросить таймер (`timer`). Следовательно, понадобятся два состояния: `timeLeft` и `timer`.

При исходной загрузке приложения таймер работать не должен; следовательно, в конструкторе `TimerWrapper` необходимо задать `timeLeft` значение `null`. Это присваивание пригодится в `Timer`, потому что вы сможете различать исходную загрузку (`timeLeft` содержит `null`) и истечение времени таймера (`timeLeft` содержит 0).

Также свойству состояния `timer` присваивается `null`. Это свойство содержит ссылку на функцию `setInterval()`, которая выполняет обратный отсчет. Но в данный момент работающего таймера нет — отсюда и значение `null`.

Наконец, выполните привязку метода `startTimer()`, потому что он будет использоваться как обработчик события (для кнопок):

```
class TimerWrapper extends React.Component {
  constructor(props) {
    super(props)
    this.state = {timeLeft: null, timer: null}
    this.startTimer = this.startTimer.bind(this)
  }
...
```

Затем идет обработчик события `startTimer`. Он вызывается каждый раз, когда пользователь щелкает по кнопке. Если пользователь щелкает по кнопке в то время, когда таймер уже работает, нужно сбросить предыдущий интервал и начать заново — ни в коем случае не должна возникнуть ситуация с несколькими одновременно работающими таймерами. По этой причине метод `startTimer()` первым делом останавливает предыдущий отсчет, вызывая метод сброса для результата вызова `setInterval()`. Объект `setInterval` текущего таймера хранится в переменной `this.state.timer`.

Для удаления результата `setInterval()` используется метод `clearInterval()`. И `clearInterval()` (http://mng.bz/7104), и `setInterval()` (http://mng.bz/P2d6) являются методами API-браузера; другими словами, они доступны из объекта окна без дополнительных библиотек и даже префиксов. (`window.clearInterval()` также сработает в браузерном коде, но не будет работать в Node.js.) Вызовите `clearInterval()` в первой строке обработчика события для кнопок:

```
class TimerWrapper extends React.Component {
  constructor(props) {
    // ...
  }
  startTimer(timeLeft) {
    clearInterval(this.state.timer)
    // ...
  }
```

После сброса предыдущего таймера можно создать новый вызовом `setInterval()`. Код, переданный `setInterval()`, будет выполняться каждую секунду. Для этого кода мы воспользуемся «стрелочной» функцией для привязки контекста `this`. Это позволит использовать состояние, свойства и методы `TimerWrapper` в этой функции метода `setInterval()`:

```
class TimerWrapper extends React.Component {
  constructor(props) {
    // ...
  }
  startTimer(timeLeft) {
    clearInterval(this.state.timer)
    let timer = setInterval(() => {
      // ...
    }, 1000)
    // ...
  }
  render() {
    // ...
  }
}
```

Теперь займемся реализацией функции. Переменная `timeLeft` определяет оставшееся время работы таймера. Мы используем ее для сохранения текущего значения, уменьшенного на 1, и проверки того, достигло ли оно 0. Если значение достигло нуля, таймер удаляется вызовом `clearInterval()` со ссылкой на объект таймера (созданный вызовом `setInterval()`), который хранится в переменной `timer`. Ссылка на таймер сохраняется в замыкании `setInterval()` даже для будущих вызовов функции (с каждой прошедшей секундой). Так работает механизм области видимости JavaScript. Следовательно, вам не нужно извлекать значение объекта `timer` из состояния (хотя при желании это возможно).

Значение `timeLeft` сохраняется в каждом цикле таймера. Наконец, `timeLeft` и объект `timer` сохраняются при щелчке по кнопке:

```
//...
startTimer(timeLeft) {
  clearInterval(this.state.timer)
  let timer = setInterval(() => {
    var timeLeft = this.state.timeLeft - 1
    if (timeLeft == 0) clearInterval(timer)
    this.setState({timeLeft: timeLeft})
  }, 1000)
  return this.setState({timeLeft: timeLeft, timer: timer})
}
//...
```

Для присваивания состояниям новых значений используется асинхронный метод `setState()`. Длина интервала `setInterval()` составляет 1000 миллисекунд, то есть 1 секунду. В состоянии нужно сохранить новые значения `timeLeft` и `timer`, потому

что приложение должно обновить эти значения, и для этого нельзя использовать простые переменные или свойства.

Вызов `setInterval()` планируется для асинхронного выполнения в цикле событий JavaScript. Возвращаемый метод `setState()` сработает перед первым обратным вызовом `setInterval()`. Вы можете легко протестировать его, включив команды консольного вывода в свой код. Например, следующий фрагмент выведет сначала 1, а потом 2, а не наоборот:

```
...
startTimer(timeLeft) {
  clearInterval(this.state.timer)
  let timer = setInterval(() => {
    console.log('2: Inside of setInterval')
    var timeLeft = this.state.timeLeft - 1
    if (timeLeft == 0) clearInterval(timer)
    this.setState({timeLeft: timeLeft})
  }, 1000)
  console.log('1: After setInterval')
  return this.setState({timeLeft: timeLeft, timer: timer})
}
...
```

Остается обязательная функция `render()` для `TimerWrapper`. Она возвращает `<h2>`, три кнопки и компонент `Timer`. `row-fluid` и `btn-group` — классы Twitter Bootstrap; они улучшают внешний вид кнопок и не критичны для React:

```
render() {
  return (
    <div className="row-fluid">
      <h2>Timer</h2>
      <div className="btn-group" role="group" >
        <Button time="5" startTimer={this.startTimer}/>
        <Button time="10" startTimer={this.startTimer}/>
        <Button time="15" startTimer={this.startTimer}/>
      </div>
```

Этот код показывает, как повторно использовать компонент `Button` с другими значениями свойства `time`. Эти значения свойства `time` позволяют кнопкам выводить разное время в своих надписях и устанавливать разные таймеры. Свойство `startTimer` компонента `Button` имеет одинаковое значение для всех трех кнопок. Используется значение `this.startTimer` из `TimerWrapper`, которое запускает/сбрасывает таймер.

Затем выводится текст «Time left: ...», который генерируется компонентом `Timer`. Для этого состояние `time` передается в свойстве компоненту `Timer`. Для соответствия рекомендованным практикам React `Timer` не имеет состояния. React автоматически обновляет текст на странице (`Timer`), когда свойство (`Timer`) обновляется изменением состояния (`TimerWrapper`). `Timer` мы реализуем позднее, а пока используйте его следующим образом:

```
<Timer time={this.state.timeLeft}/>
```

Кроме того, тег `<audio>` (тег HTML5, содержащий ссылку на файл) предупредит вас об истечении времени:

```
        <audio id="end-of-time" src="flute_c_long_01.wav" preload="auto">
        ➥ </audio>
      </div>
    )
  }
}
```

Для вашего удобства и лучшего понимания (иногда бывает полезно видеть весь компонент сразу) я приведу полный код `TimerWrapper` (ch11/timer/timer.jsx).

Листинг 11.4. Компонент TimerWrapper

```
class TimerWrapper extends React.Component {
  constructor(props) {
    super(props)
    this.state = {timeLeft: null, timer: null}
    this.startTimer = this.startTimer.bind(this)
  }
  startTimer(timeLeft) {
    clearInterval(this.state.timer)
    let timer = setInterval(() => {
      console.log('2: Inside of setInterval')
      var timeLeft = this.state.timeLeft - 1
      if (timeLeft == 0) clearInterval(timer)
      this.setState({timeLeft: timeLeft})
    }, 1000)
    console.log('1: After setInterval')
    return this.setState({timeLeft: timeLeft, timer: timer})
  }
  render() {
    return (
      <div className="row-fluid">
        <h2>Timer</h2>
        <div className="btn-group" role="group" >
          <Button time="5" startTimer={this.startTimer}/>
          <Button time="10" startTimer={this.startTimer}/>
          <Button time="15" startTimer={this.startTimer}/>
        </div>
        <Timer timeLeft={this.state.timeLeft}/>
        <audio id="end-of-time" src="flute_c_long_01.wav"
        ➥ preload="auto"></audio>
      </div>
    )
  }
}
```

Сбрасывает таймер на случай, если работают другие таймеры

Обновляет уменьшенное время каждую секунду

Генерирует кнопки для вызова startTimer с разным временем

Тег HTML5 <audio> воспроизводит сигнал при достижении 0

Генерирует текст «Time left:...» и выводит звуковой сигнал при достижении 0

`TimerWrapper` содержит значительный объем логики. Другие компоненты не обладают состоянием и не блещут функциональностью. Впрочем, их все равно нужно реализовать. Помните тег `<audio>` в `TimerWrapper`, который воспроизводит звуки, когда оставшееся время достигает 0? Перейдем к компоненту `Timer`.

11.4. Компонент Timer

Компонент `Timer` должен выводить оставшееся время и воспроизводить звуковой сигнал при его истечении. Этот компонент не обладает состоянием. Реализуйте класс и организуйте проверку того, достигло ли свойство `timeLeft` нуля:

```
class Timer extends React.Component {
  render() {
    if (this.props.timeLeft == 0) {
      // ...
    }
      // ...
  }
}
```

Для воспроизведения звука (файл flute_c_long_01.wav) в этом проекте используется специальный элемент HTML5 `<audio>`; вы определили его в `TimerWrapper`, при этом атрибут `src` содержит ссылку на WAV-файл, а `id` присваивается значение `end-of-time`. Все, что нужно сделать, — получить узел DOM (стандартная функция JavaScript `getElementById()` прекрасно сработает) и вызвать `play()` (также стандартная функция JavaScript, начиная с HTML5). Этот момент еще раз показывает, как хорошо React сочетается с другими технологиями, связанными с JavaScript, — HTML5, jQuery 3[1] и даже Angular 4, если вам хватит смелости:

```
class Timer extends React.Component {
  render() {
    if (this.props.timeLeft == 0) {
      document.getElementById('end-of-time').play()
    }
    // ...
```

Как объяснялось ранее, в исходном состоянии на таймере не должен отображаться текст «0», потому что таймер еще не запускался. По этой причине в `TimerWrapper` (листинг 11.4) `timeLeft` присваивается исходное значение `null`. Если `timeLeft` равно `null` или 0, то компонент `Timer` генерирует пустой элемент `<div>`. Иначе говоря, в приложении не будет выводиться 0:

[1] За примерами интеграции с событиями браузера и jQuery обращайтесь к главе 3.

```
if (this.props.timeLeft == null || this.props.timeLeft == 0)
  return <div/>
```

В противном случае, когда `timeLeft` больше 0, в элементе `<h1>` выводится оставшееся время. А это означает, что оставшееся время должно отображаться во время работы таймера:

```
return <h1>Time left: {this.props.timeLeft}</h1>
```

В листинге 11.5 приведен полный код компонента `Timer` (ch11/timer/timer.jsx).

Листинг 11.5. Компонент Timer с оставшимся временем

```
class Timer extends React.Component {
  render() {
    if (this.props.timeLeft == 0) {
      document.getElementById('end-of-time').play()
    }
    if (this.props.timeLeft == null || this.props.timeLeft == 0)
      return <div/>
    return <h1>Time left: {this.props.timeLeft}</h1>
  }
}
```

Воспроизводит звуковой сигнал по истечении времени

В исходном состоянии не выводится ничего

Выводит текст «Time left:…»

Чтобы компонент `Timer` выводил количество секунд, таймер сначала нужно запустить, а это происходит по щелчку на кнопках. Так займемся же кнопками!

11.5. Компонент Button

В соответствии с принципом DRY[1] (Don't Repeat Yourself), вы создаете один компонент `Button` и трижды используете его для вывода трех разных кнопок. `Button` — еще один компонент без состояния (и притом очень простой), как того требует менталитет React, но компонент `Button` все же сложнее `Timer`, потому что у него есть обработчик события.

Кнопки должны иметь обработчик события `onClick` для получения щелчков, сделанных пользователем. Щелчки должны запускать обратный отсчет времени. Функция запуска таймера не реализована в `Button`: она реализована в `TimerWrapper` и передается компоненту `Button` от родителя `TimerWrapper` в `this.`

[1] Принцип DRY определяется следующим образом: «Каждый фрагмент знаний должен иметь единственное, однозначное, общепризнанное представление в системе»; см. «Don't Repeat Yourself», Википедия, http://mng.bz/1K5k; и «The Pragmatic Programmer: From Journeyman to Master» Эндрю Ханта (Andrew Hunt) (Addison-Wesley Professional, 1999), http://amzn.to/2ojjXoY.

`props.startTimer`. Но как передать время (`5`, `10` или `15`) функции `startTimer` из `TimerWrapper`? Взгляните на следующий код из `TimerWrapper`, который передает периоды времени в свойствах:

```
<Button time="5" startTimer={this.startTimer}/>
<Button time="10" startTimer={this.startTimer}/>
<Button time="15" startTimer={this.startTimer}/>
```

Идея заключается в том, чтобы выполнить рендер трех кнопок с использованием этого компонента (повторное использование кода — ура!). Но чтобы узнать, какое время было выбрано пользователем, вам понадобится значение из `this.props.time`, которое передается в аргументе `this.props.startTimer`.

Например, следующий код работать не будет:

```
// Работать не будет - необходимо определение.
<button type="button" className='btn-default btn'
  onClick={this.props.startTimer(this.props.time)}>
  {this.props.time} seconds
</button>
```

Функция, переданная `onClick`, должна быть определением, а не вызовом. Как насчет такого варианта?

```
// Да. Вот теперь вы на правильном пути.
<button type="button" className='btn-default btn'
  onClick={()=>{this.props.startTimer(this.props.time)}}>
  {this.props.time} seconds
</button>
```

Да, в этом фрагменте содержится правильный код передачи значения: на среднем шаге (функция) передаются разные значения времени. Чтобы решение было более элегантным, можно создать метод класса. Другой возможный вариант — замена промежуточной функции на каррирующий[1] вызов `bind()`:

```
onClick = {this.props.startTimer.bind(null, this.props.time)}
```

Вспомните, что `bind()` возвращает определение функции. Пока `onClick` (или любому другому обработчику события) передается определение функции, все хорошо.

Вернемся к компоненту `Button`. Обработчик события `onClick` вызывает метод класса `this.startTimer`, который, в свою очередь, вызывает функцию из свойства `this.props.startTimer`. Вы можете использовать этот объект (`this.props.startTimer`) в `this.startTimer` благодаря вызову `bind(this)`.

[1] https://ru.wikipedia.org/wiki/Каррирование. — *Примеч. пер.*

Компонент `Button` не имеет состояния, в чем нетрудно убедиться, просмотрев его полный код в листинге 11.6 (ch11/timer/timer.jsx). Что это означает? То, что его можно переработать в функцию вместо класса.

Листинг 11.6. Компонент Button, запускающий обратный отсчет

```
class Button extends React.Component {
  startTimer(event) {  ◄—— Запускает или сбрасывает таймер с правильным значением времени
    return this.props.startTimer(this.props.time)
  }
  render() {  ◄—— Рендерит пользовательский интерфейс Button
    return <button type="button" className='btn-default btn'
      onClick={this.startTimer.bind(this)}>  ◄—— Перехватывает onClick
        {this.props.time} seconds
    </button>
  }
}
```

Разумеется, вы не обязаны использовать одинаковые имена методов (такие, как `startTimer()`) в `Button` и `TimerWrapper`. Многие люди путаются на моих семинарах React, когда я использую одинаковые имена; другим с одинаковыми именами проще отслеживать цепочки вызовов. Просто знайте, что методу `Button` можно присвоить другое имя, например `handleStartTimer()`. Лично я считаю, что использование того же имени помогает создать мысленную связь между свойствами, методами и состояниями внутренних компонентов.

Кроме того, `Timer` также можно было бы назвать `TimerLabel` — хотя бы для метода воспроизведения звука `play()`. Есть ли другие возможности для усовершенствования и рефакторинга? Безусловно! Обратитесь к разделу «Домашнее задание» этой главы.

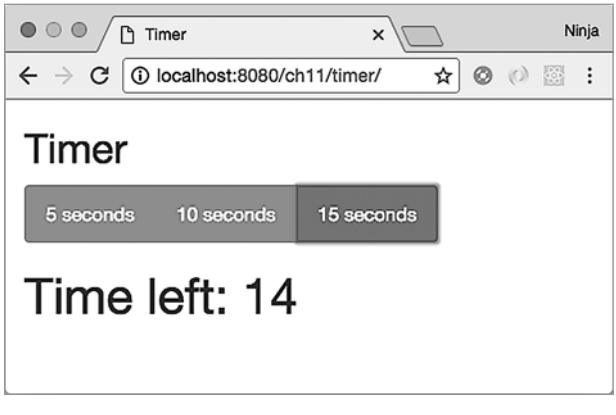


Рис. 11.4. Таймер был запущен кнопкой 15 seconds. Теперь таймер показывает, что осталось 14 секунд

Поздравляю — программирование на этом можно считать завершенным. Теперь нужно запустить результат, чтобы вы могли использовать этот таймер для работы[1] или увлечений.

11.6. Запуск таймера

Откомпилируйте JSX в JavaScript следующей командой Babel 6.9.5 (предполагается, что Babel CLI и его конфигурации уже установлены; подсказка — package.json!):

```
$ ./node_modules/.bin/babel timer.jsx -o timer.js -w
```

Если вы скопировали мой сценарий сборки npm из package.json в начале этой главы, можно выполнить команду `npm run build`. Если все было сделано правильно, на экране появится красивый таймер, показанный на рис. 11.4. Отключите музыку, чтобы услышать сигнал по истечении времени.

Убедитесь в том, что приложение работает *правильно*: оставшееся время должно уменьшаться с каждой секундой. Нажатие кнопки должно запускать новый отсчет времени; иначе говоря, таймер прерывает работу и перезапускается при каждом нажатии кнопки.

11.7. Домашнее задание

Для получения дополнительных баллов сделайте следующее:

- Преобразуйте `Timer` в компонент без состояния, реализованный «стрелочной» функцией.
- Реализуйте кнопку Pause/Resume для остановки/возобновления работы таймера.
- Реализуйте кнопку Cancel, которая останавливает отсчет и скрывает оставшееся время.
- Реализуйте кнопку Reset для сброса оставшегося времени и возврата к исходному значению (5, 10 или 15 секунд).
- Измените итоговую версию проекта так, чтобы в ней использовались интервалы 5, 10 или 15 минут (вместо секунд).
- Отделите тег `<audio>` в `TimerWrapper` от `play()` в `Timer`.

[1] Опробуйте методику Pomodoro (https://cirillocompany.de/pages/pomodoro-technique) для повышения вашей эффективности.

- Проведите рефакторинг проекта, чтобы он содержал четыре файла — timer.jsx, timer-label.jsx, timer-button.jsx и timer-sound.jsx, с максимумом слабых связей.
- Реализуйте кнопку ползунка, которая изменяется с каждым прошедшим интервалом (интеграция с ползунком рассматривается в главе 6).

Отправьте свой код в новую папку в ch11 как pull-запрос в репозиторий GitHub этой книги: https://github.com/azat-co/react-quickly.

11.8. Итоги

- Старайтесь делать свои компоненты как можно более простыми и близкими к презентационным.
- Передавайте функции как значения свойств, а не как обычные данные.
- Два компонента могут обмениваться данными друг с другом через родителя.

ЧАСТЬ 2

Архитектура React

Приветствую вас в части 2! После знакомства с важнейшими концепциями, функциональными возможностями и паттернами React вы готовы отправиться в самостоятельное плавание. Часть 1 подготовила вас к построению простых UI-элементов; и если вы занимаетесь построением веб-интерфейсов, базовых возможностей React будет достаточно. Но для построения полноценных приложений клиентской части разработчики React используют модули с открытым кодом, написанные сообществом React. Большинство таких модулей размещается на GitHub и npm, и они находятся у вас под рукой — просто берите и начинайте использовать.

В этих главах рассматриваются самые популярные, наиболее часто используемые, проверенные временем библиотеки, которые в сочетании с базовой функциональностью React образуют стек React (или React с друзьями, как некоторые разработчики шуточно называют эту комбинацию). Для начала в главах 12–17 вы узнаете об использовании Webpack для конвейеров активов, React Router — для маршрутизации URL, Redux и GraphQL — для работы с данными, Jest — для тестирования, Express и Node — для Universal React. Затем, как и в части 1, в главах 18–20 представлены реальные проекты.

На первый взгляд вам предстоит значительная работа, но мой опыт чтения и написания книг показал, что «первые шаги» и учебные примеры не несут особой ценности для читателей и не показывают, как всё работает в реальной жизни. Поэтому в этой части книги вы узнаете о стеке React и научитесь работать с ним. Вас ожидают интересные, сложные проекты. После завершения этой части вы будете разбираться в работе с данными, освоите настройку колосса, называемого Webpack, и сможете уверенно обсуждать эти темы на местных встречах.

Продолжайте читать.

12 Система сборки Webpack

Посмотрите вступительный видеоролик к этой главе, отсканировав QR-код или перейдя по ссылке http://reactquickly.co/videos/ch12.

ЭТА ГЛАВА ОХВАТЫВАЕТ СЛЕДУЮЩИЕ ТЕМЫ:

- Включение Webpack в проект.
- Модуляризация кода.
- Запуск Webpack и тестирование сборки.
- Оперативная замена модулей.

Прежде чем браться за изучение стека React, рассмотрим инструмент, чрезвычайно важный для многих современных проектов веб-разработки: он называется *системой сборки* (или *сборщиком*). Этот инструмент будет использоваться в последующих главах для сборки множества файлов с кодом в минимальное число файлов, необходимых для запуска приложений и их подготовки к простому развертыванию. Мы будем использовать систему сборки Webpack (https://webpack.js.org).

Если ранее вы еще не сталкивались с системами сборки или работали с другой системой (например, Grunt, Gulp или Bower), то эта глава написана для вас. Вы узнаете, как установить Webpack, настроить и использовать с проектом. Также в этой главе рассматривается оперативная замена модулей (HMR, Hot Module Replacement) — возможность Webpack, позволяющая оперативно заменять обновленные модули теми, которые работают на действующем сервере. Однако сначала посмотрим, что вам может предложить Webpack.

ПРИМЕЧАНИЕ Генераторы кода, такие как `create-react-app` (https://github.com/facebookincubator/create-react-app), создают шаблонный/инициализационный код и помогают

быстро создавать проекты. `create-react-app` также использует Webpack и Babel наряду с другими модулями. Однако в этой книге прежде всего излагаются основы, и генератор кода нам не понадобится; вместо этого мы проведем все подготовительные действия вручную, чтобы вы хорошо поняли каждую часть. Если вас заинтересует эта тема, вы можете освоить работу с генератором кода самостоятельно — все сводится к нескольким командам.

ПРИМЕЧАНИЕ Исходный код примеров этой главы доступен по адресам www.manning.com/books/react-quickly и https://github.com/azat-co/react-quickly/tree/master/ch12 (в папке ch12 репозитория GitHub https://github.com/azat-co/react-quickly). Также некоторые демонстрационные примеры доступны по адресу http://reactquickly.co/demos.

12.1. Что делает Webpack?

Вы когда-нибудь задавались вопросом, почему (в области веб-разработки) каждый встречный говорит о Webpack? Главная задача Webpack — оптимизация написанного вами кода JavaScript, чтобы он содержал как можно меньше файлов, которые будут запрашиваться клиентом. Такой подход сокращает нагрузку на серверы для популярных сайтов, а также сокращает время загрузки страницы клиентом. Конечно, все не так просто. Код JavaScript часто оформляется в модули, подходящие для повторного использования. Но эти модули часто зависят от других модулей, которые могут зависеть от третьих модулей, и т. д.; отслеживание всех необходимых загрузок при таком количестве зависимостей быстро превращается в головную боль.

Допустим, у вас имеется вспомогательный модуль `myUtil`, который используется во многих компонентах React — accounts.jsx, transactions.jsx и т. д. Без такого инструмента, как Webpack, вам придется вручную следить за тем, чтобы при каждом использовании одного из этих компонентов модуль `myUtil` включался как зависимость. Кроме того, это может привести к избыточным загрузкам `myUtil` по второму и третьему разу, потому что он мог быть уже загружен ранее другим компонентом, зависящим от `myUtil`. Конечно, это сильно упрощенный пример; в реальных проектах могут присутствовать десятки и даже сотни зависимостей, используемых в других зависимостях. Webpack поможет вам в управлении ими.

Webpack умеет работать со всеми тремя типами модулей JavaScript: CommonJS (www.commonjs.org), AMD (https://github.com/amdjs/amdjs-api/wiki/AMD) и ES6 (http://mng.bz/VjyO), так что вам не нужно беспокоиться, если вы работаете с произвольной комбинацией разнотипных модулей. Webpack проанализирует зависимости для всего кода JavaScript в вашем проекте, а затем проследит за тем, чтобы:

- все зависимости загружались в правильном порядке;
- все зависимости загружались только один раз;
- код JavaScript был упакован в наименьшее количество файлов (называемых *ассетами* — static assets).

Webpack также поддерживает *разделение кода* и *хеширование асссетов* для выявления блоков кода, которые необходимы только в определенных обстоятельствах. Эти блоки отделяются от остального кода для загрузки по требованию, вместо того чтобы собираться со всем остальным. Вы должны явно запросить использование этих возможностей для дальнейшей оптимизации своего кода JavaScript и его развертывания.

> **ПРИМЕЧАНИЕ** Разделение кода и хеширование ассетов выходит за рамки этой книги. За дополнительной информацией обращайтесь на сайт Webpack: https://webpack.github.io/docs/code-splitting.html.

Впрочем, Webpack не ограничивается кодом JavaScript. Также поддерживается предварительная обработка других статических файлов посредством использования загрузчиков. Например, перед сборкой вы можете:

- Выполнить предварительную компиляцию файлов JSX, Jade или CoffeeScript в обычный код JavaScript.
- Выполнить предварительную компиляцию кода ES6+ в ES5 для браузеров, которые пока не поддерживают ES6.
- Выполнить предварительную компиляцию файлов Sass и Compass в CSS.
- Оптимизировать двумерную графику в один файл PNP, файл JPG или встроенные ассеты.

Существует множество загрузчиков для разных типов файлов. Кроме того, на домашней странице Webpack имеется каталог плагинов, изменяющих поведение Webpack. Если вы не сможете найти то, что нужно, имеется документация по написанию собственных плагинов.

В оставшейся части книги мы будем использовать Webpack для выполнения следующих операций:

- Управление зависимостями и их упаковка из модулей npm, чтобы вам не приходилось вручную загружать файлы из интернета и включать их при помощи тегов `<script>` в HTML.
- Транспиляция JSX в обычный код JavaScript с картами исходного кода для упрощения отладки.
- Управление стилями.
- Выполнение оперативной перезагрузки модулей.
- Построение веб-сервера для разработки.

Как будет показано ниже, вы можете настроить порядок, в котором Webpack загружает, осуществляет предварительную компиляцию и упаковывает файлы, с использованием файла webpack.config.js. Но сначала посмотрим, как установить систему Webpack и обеспечить ее работу с проектом.

12.2. Включение Webpack в проект

Чтобы показать, как начать работу с Webpack, мы слегка изменим проект из главы 7 (рис. 12.1). На форме присутствуют поля для ввода адреса электронной почты и комментария, проект содержит две таблицы стилей и один компонент `Content`.

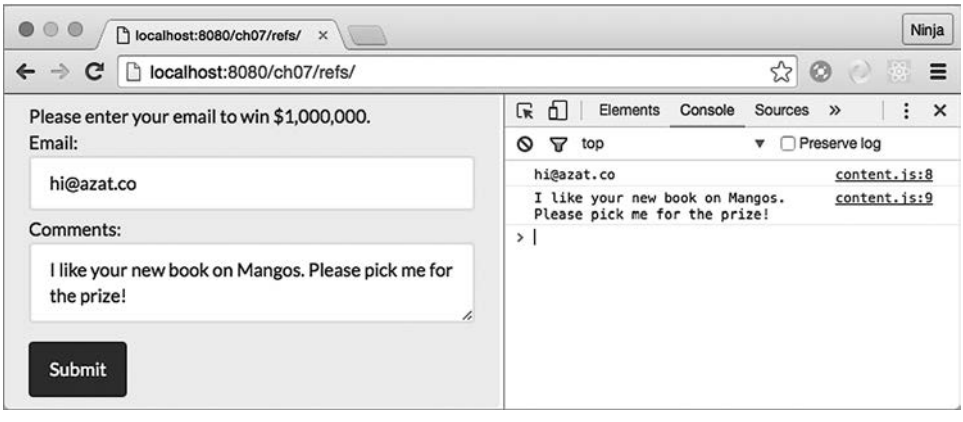


Рис. 12.1. Исходный проект перед использованием Webpack

Ниже показана новая структура проекта. Я выделил ее отличия от проекта из главы 7:

```
/email-webpack
  /css
    bootstrap.css
    main.css
  /js  ◄—— Не содержит файлы react.js или react-dom.js
    bundle.js  ◄—— Все сценарии
    bundle.map.js  ◄—— Соответствия номеров строк для DevTools
  /jsx
    app.jsx  ◄—— Команда ReactDOM.render
    content.jsx
  /node_modules  ◄—— Зависимости для компиляции (Webpack, Babel и т. д.)
  index.html
  package.json  ◄—— Данные конфигурации Babel и другая информация о проекте
  webpack.config.js  ◄—— Данные конфигурации Webpack
  webpack.dev-cli.config.js
  webpack.dev.config.js
```

Сравните со структурой без использования Webpack из главы 7:

```
/email
  /css
    bootstrap.css
  /js
```

```
  content.js  ◄──  Скомпилированный сценарий с главным компонентом
  react.js
  react-dom.js
  script.js
/jsx
  content.jsx
  script.jsx  ◄──  Команда ReactDOM.render() в JSX
index.html
```

ПРИМЕЧАНИЕ А у вас установлены Node.js и npm? Сейчас самое время установить их — они понадобятся вам для продолжения работы. Процесс установки описан в приложении А.

В этом разделе рассматриваются следующие темы:

1. Установка Webpack.
2. Установка зависимостей и сохранение их в package.json.
3. Настройка конфигурации Webpack в файле webpack.config.js.
4. Настройка сервера разработки и оперативной замены модулей.

Итак, начнем.

12.2.1. Установка системы Webpack и ее зависимостей

Для использования Webpack необходимо несколько дополнительных зависимостей, упомянутых в package.json:

- *Webpack* — инструмент сборки (имя в npm: `webpack`); используйте версию 2.4.1.
- *Загрузчики* — стили, CSS, оперативная замена модулей (HMR) и препроцессоры Babel/JSX (имена в npm: `style-loader`, `css-loader`, `react-hot-loader` и `babel-loader`, `babel-core` и `babel-preset-react`); используйте версии, указанные в package.json.
- *webpack-dev-server* — сервер разработки Express, который позволяет использовать HMR (имя в npm: `webpack-dev-server`); используйте версию 2.4.2.

Каждый модуль можно установить вручную, но я рекомендую скопировать файл package.json в листинге 12.1 (ch12/email-webpack/package.json) из репозитория GitHub в корневой каталог вашего проекта (структура проекта приведена в разделе 12.2). Затем выполните команду `npm i` или `npm install` из корневого каталога проекта (где находится package.json) для установки зависимостей. Команда проследит за тем, чтобы вы не забыли ни один из 10 модулей (синоним для пакета в Node). Она также гарантирует, что ваши версии будут близки к тем, которые использовались мной. Значительные расхождения в версиях — *замечательный* способ нарушить работу приложения!

Листинг 12.1. Подготовка среды разработки

```
{
  "name": "email-webpack",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "build": "./node_modules/.bin/webpack -w"
    "wds-cli": "./node_modules/.bin/webpack-dev-server --inline --hot
     ➥ --module-bind 'css=style-loader!css-loader'
     ➥ --module-bind 'jsx=react-hot-loader!babel-loader'
     ➥ --config webpack.dev-cli.config.js",
     "wds": "./node_modules/.bin/webpack-dev-server --config
     ➥ webpack.dev.config.js"
  },
  "author": "Azat Mardan",
  "license": "MIT",
  "babel": {
    "presets": [
      "react"
    ]
  },
  "devDependencies": {
    "babel-core": "6.13.2",
    "babel-loader": "6.4.1",
    "babel-preset-react": "6.5.0",
    "css-loader": "0.23.1",
    "react": "15.5.4",
    "react-dom": "15.5.4",
    "react-hot-loader": "1.3.1",
    "style-loader": "0.13.1",
    "webpack": "2.4.1",
    "webpack-dev-server": "2.4.2"
  },
}
```

Сохраняет сценарий сборки Webpack как сценарий npm

Сообщает Babel, какие конфигурации следует использовать (React для JSX в данном случае; конфигурация ES6+ необязательна)

Устанавливает загрузчик Babel для обработки JSX

Устанавливает загрузчик CSS для запроса CSS из JavaScript, а затем устанавливает загрузчик Style для внедрения CSS в веб-страницу

Устанавливает загрузчик React HMR

Устанавливает Webpack локально (рекомендуется)

Устанавливает webpack-dev-server локально (рекомендуется)

Свойство `babel` в файле package.json уже знакомо вам по части 1 этой книги, поэтому я не буду повторяться. Напомню лишь, что это свойство необходимо для настройки Babel, чтобы преобразовывать JSX в JS. Если вам нужно обеспечить поддержку браузеров, не работающих с ES6, включите конфигурацию `es2015` в раздел `presets`:

```
"babel": {
  "presets": [
    "react",
    "es2015"
  ]
},
```

Также добавьте `babel-preset-es2015` в раздел `devDependencies`:

```
"devDependencies": {
  "babel-preset-es2015": "6.18.0",
  ...
}
```

Кроме новых зависимостей, также имеются сценарии npm. Использовать команды сценариев в package.json не обязательно, но весьма желательно, потому что применение сценариев npm для запуска и сборки считается наиболее эффективной практикой при работе с React и Node. Конечно, вы можете проводить всю сборку вручную без сценариев npm, но зачем набирать столько лишнего текста?

Webpack можно запустить либо командой `npm run buid`, либо напрямую командой `./node_modules/.bin/webpack -w`. Флаг `-w` означает отслеживание изменений (watch), то есть постоянное наблюдение за любыми изменениями в исходном коде и повторная сборка пакетов при их обнаружении. Иначе говоря, Webpack продолжит работу для автоматического внесения изменений. Конечно, все необходимые модули должны быть установлены командой `npm i`.

Команда `webpack -w` по умолчанию ищет файл webpack.config.js. Без этого конфигурационного файла Webpack работать не будет. Сейчас мы займемся его созданием.

> **ПРИМЕЧАНИЕ** Сценарии `wds` и `wds-cli` из package.json рассматриваются в разделе 12.5.

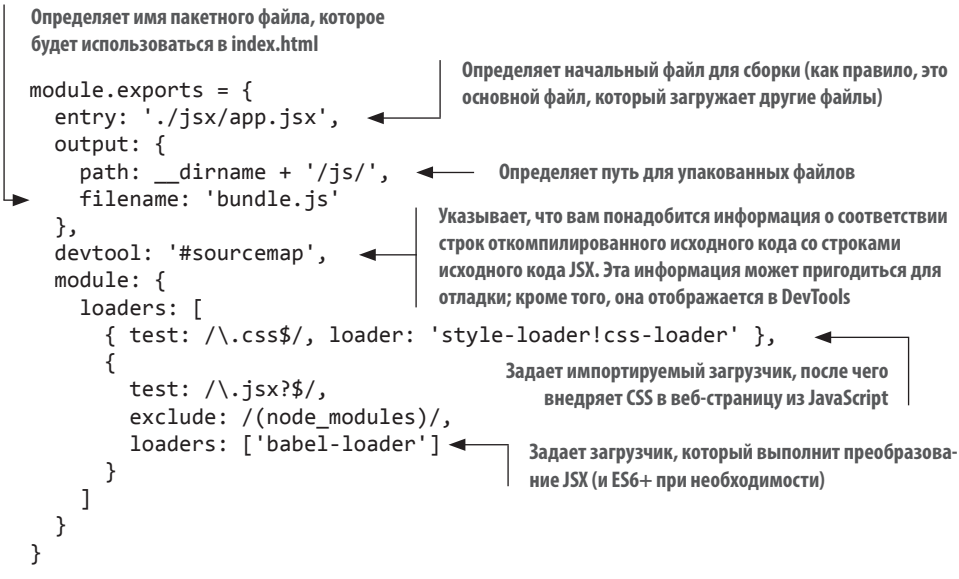
12.2.2. Настройка Webpack

Webpack необходимо знать, что нужно обрабатывать (исходный код) и как это делать (с загрузчиками). Именно поэтому файл webpack.config.js находится в корне структуры проекта. Если вкратце, в этом проекте Webpack используется для решения следующих задач:

- Преобразование файлов JSX в файлы JS: `babel-loader`, `babel-core` и `babel-preset-react`.
- Загрузка CSS с использованием `require`, преобразование `url` и `imports` в процессе с использованием `css-loader` (https://github.com/webpack/css-loader).
- Добавление CSS внедрением элемента `<style>` с использованием `style-loader` (https://github.com/webpack/style-loader).
- Сборка всех полученных файлов JS в один файл с именем bundle.js.
- Передача DevTools соответствий строк исходного кода с использованием карт исходного кода.

Webpack необходим собственный конфигурационный файл: email-webpack/webpack.config.js.

Листинг 12.2. Конфигурационный файл Webpack




```
module.exports = {
  entry: './jsx/app.jsx',
  output: {
    path: __dirname + '/js/',
    filename: 'bundle.js'
  },
  devtool: '#sourcemap',
  module: {
    loaders: [
      { test: /\.css$/, loader: 'style-loader!css-loader' },
      {
        test: /\.jsx?$/,
        exclude: /(node_modules)/,
        loaders: ['babel-loader']
      }
    ]
  }
}
```

Свойство `devtool` полезно на стадии разработки, потому что оно предоставляет карты исходного кода, содержащие номера строк в исходном — неоткомпилированном — коде. Теперь вы готовы к запуску Webpack для этого проекта, а также к самостоятельной настройке любых проектов на базе Webpack в будущем.

КОНФИГУРАЦИОННЫЕ ФАЙЛЫ

При желании можно создать сразу несколько конфигурационных файлов — для разработки, реальной эксплуатации, тестирования и других вариантов сборки. В структуре проекта текущего примера я создал следующие файлы:

```
webpack.dev-cli.config.js
webpack.dev.config.js
```

Имена не имеют значения — важно лишь то, чтобы вы и ваши коллеги понимали назначение каждого файла. Имя передается Webpack с ключом --config. Эти конфигурационные файлы более подробно рассматриваются в разделе 12.4.

Webpack предоставляет массу полезных возможностей; мы рассмотрели лишь основные из них, но и этого достаточно для компиляции JSX, создания карт исходного кода, внедрения и импортирования CSS, а также упаковки JavaScript. Когда вам потребуется расширенная функциональность Webpack, обратитесь к документации или книге — например, «SurviveJS» Юхо Вепсалайнена (Juho Vepsäläinen) (https://survivejs.com).

Итак, теперь вы готовы воспользоваться мощью Webpack в JSX.

12.3. Модуляризация кода

Напомню, что приложение из главы 7 использовало глобальные объекты и `<script>`. Это нормально подходит для такой книги или небольшого приложения. Однако в крупных приложениях использовать глобальные объекты не рекомендуется, потому что у вас могут возникнуть проблемы с конфликтами имен или управлением тегами `<script>` с повторными включениями. Вы можете поручить все управление зависимостями Webpack при помощи синтаксиса CommonJS. Webpack включает только необходимые зависимости и упаковывает их в один файл bundle.js (на основании содержимого webpack.config.js).

Организация кода посредством разбиения его на модули желательна не только для React, но и для любого программного проекта вообще. Вы можете воспользоваться Browserify, SystemJS или любым другим сборщиком/загрузчиком модулей, и все равно использовать синтаксис CommonJS/Node.js (`require` и `module.exports`). Таким образом, код в этом разделе можно будет перенести в другие системы после того, как вы избавите его от примитивных глобальных объектов посредством рефакторинга.

На момент написания книги конструкция `import` (http://mng.bz/VjyO) поддерживается только одним браузером — Edge и не поддерживается Node.js. Модули ES6 с синтаксисом `import` потребуют дополнительной работы при настройке Webpack. Этот синтаксис не является точной заменой для синтаксиса CommonJS `require`/`module.exports`, поскольку эти команды работают по-разному. По этой причине в листинге 12.3 (ch12/email-webpack/app.jsx) код app.jsx переработан для использования `require()` и `module.exports` вместо глобальных объектов и тега HTML `<script>`. Благодаря использованию `style-loader` вы также можете использовать `require` с файлами CSS, а благодаря загрузчику Babel — использовать `require` с файлами JSX.

Листинг 12.3. Рефакторинг app.jsx

```
require('../css/main.css')  <-- Импортирует код CSS, который благодаря загрузчикам стилей и css будет импортирован и внедрен в веб-страницу

const React = require('react')  <-- Импортирует React для синтаксиса <>: React.createElement()
const ReactDOM = require ('react-dom')
const Content = require('./content.jsx')  <-- Импортирует Content

ReactDOM.render(
  <Content />,
  document.getElementById('content')
)
```

Сравните с файлом ch07/email/jsx/script.jsx:

```
ReactDOM.render(
  <Content />,
  document.getElementById('content')
)
```

Старый файл был меньше, но это один из тех редких случаев, когда «меньше» не значит «лучше». В этом файле использовались глобальные объекты `Content`, `ReactDOM` и `React`, а это, как я только что объяснил, считается нежелательной практикой.

В файле content.jsx можно использовать `require()` похожим образом. Код `constructor()`, `submit()` и `render()` не изменился:

```
const React = require('react')  ◄——  Импортирует React
const ReactDOM = require('react-dom')  ◄——  Импортирует ReactDOM

class Content extends React.Component {
  constructor(props) {
      // ...
  }
  submit(event) {
      // ...
  }
  render() {
      // ...
  }
}

module.exports = Content  ◄——  Экспортирует Content
```

Файл index.html должен содержать ссылку на пакет, который система Webpack создала для вас: js/bundle.js. Его имя указано в webpack.config.js, и теперь его нужно добавить. Он будет создан после выполнения команды `npm run build`. Новая разметка index.html выглядит так:

```
<!DOCTYPE html>
<html>

  <head>
    <link href="css/bootstrap.css" type="text/css" rel="stylesheet"/>
  </head>

  <body>
    <div id="content" class="container"></div>
    <script src="js/bundle.js"></script>
  </body>

</html>
```

Также из файла index.html была удалена ссылка на таблицу стилей main.css. Webpack внедрит элемент `<style>` со ссылкой на main.css в index.html благодаря команде `require('main.css')` в app.jsx. Также `require()` можно использовать для bootstrap.css.

И это был последний этап рефакторинга вашего проекта.

12.4. Запуск Webpack и тестирование сборки

Наступает момент истины. Выполните команду `$ npm run build` и сравните свой результат со следующим:

```
> email-webpack@1.0.0 build
➥ /Users/azat/Documents/Code/react-quickly/ch12/email-webpack
> webpack -w

Hash: 2ffe09fff88a4467788a
Version: webpack 1.12.9
Time: 2545ms
        Asset      Size   Chunks             Chunk Names
    bundle.js    752 kB        0  [emitted]   main
bundle.js.map    879 kB        0  [emitted]   main
    + 177 hidden modules
```

Если все прошло без ошибок и вы видите только что созданные файлы bundle.js и bundle.js.map в папке js — браво! Теперь запустите свой любимый веб-сервер (возможно, node-static или http-server) и протестируйте веб-приложение. Вы увидите, что адреса электронной почты и комментарии выводятся в консоль.

Как видите, интеграция Webpack с проектом выполняется достаточно прямолинейно и дает замечательный результат.

---

177 СКРЫТЫХ МОДУЛЕЙ, ИЛИ ПАКЕТ WEBPACK ИЗНУТРИ

Файл ch12/email-webpack/js/bundle.js содержит 177 модулей! Вы можете открыть файл и найти в нем команды webpack_require(1), webpack_require(2) и т. д. до команды webpack_require(176) для компонента Content. Следующий откомпилированный код из app.jsx импортирует Content (строки 49–53 из bundle.js):

```
const React = __webpack_require__(5);
const ReactDOM = __webpack_require__(38);
const Content = __webpack_require__(176);

ReactDOM.render(React.createElement(Content, null),
➥ document.getElementById('content'));
```

Как минимум вы готовы к использованию Webpack в остальных примерах книги. Но я настоятельно рекомендую настроить еще одну возможность: оперативную замену модулей (HMR), которая может кардинально ускорить разработку. Прежде чем переходить к разработке React, рассмотрим эту замечательную возможность Webpack.

ESLINT И FLOW

Я хочу упомянуть еще два полезных инструмента разработки. Разумеется, они не обязательны, но могут принести огромную пользу.

ESLint (http://eslint.org, в `npm eslint`) получает заранее определенные правила или наборы правил и проверяет, соответствует ли ваш код (JS или JSX) тем же стандартам. Например, сколько пробелов содержит один отступ — два или четыре? А если вы случайно поставили точку с запятой в своем коде? (В JavaScript символы «точка с запятой» необязательны, и я предпочитаю обходиться без них.) ESLint даже выдаст предупреждение о неиспользуемых переменных. А это может предотвратить проникновение в код ошибок (не всех, конечно)!

За информацией обращайтесь к статье «Getting Started with ESLint» (http://eslint.org/docs/user-guide/getting-started). Также вам понадобится `eslint-plugin-react` (https://github.com/yannickcr/eslint-plugin-react). Не забудьте добавить правила React в .eslintrc.json (полный код находится в папке ch12/email-webpack-eslint-flow):

```
"rules": {
    "react/jsx-uses-react": "error",
    "react/jsx-uses-vars": "error",
  }
```

Пример предупреждений, выдаваемых при выполнении ESLint React для ch12/email-web-pack-lint-flow/jsx/content.jsx:

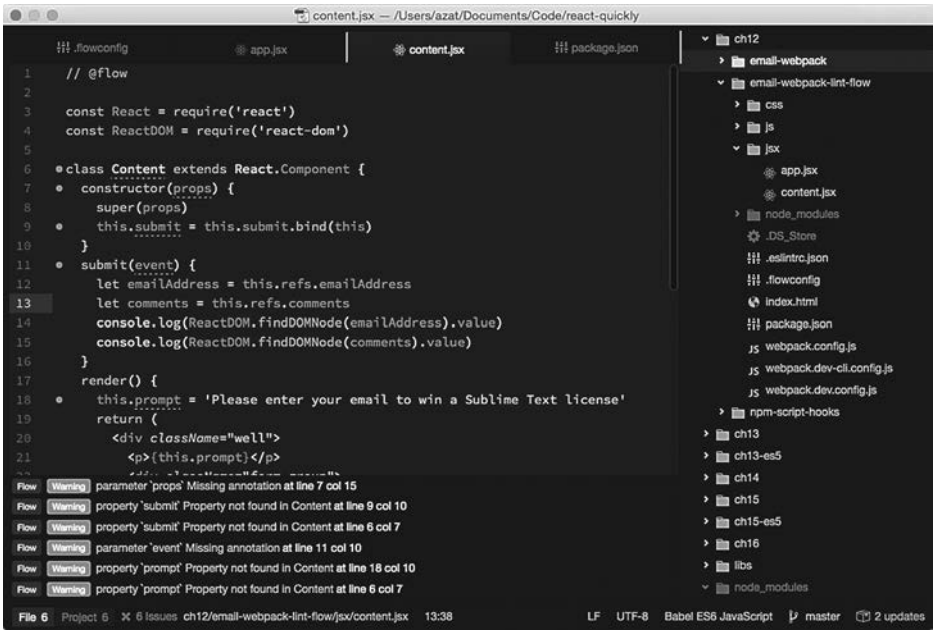
```
/Users/azat/Documents/Code/react-quickly/ch12/
  ↪ email-webpack-lint-flow/jsx/content.jsx
  9:10   error  'event' is defined but never used  no-unused-vars
  12:5   error  Unexpected console statement       no-console
  12:17  error  Do not use findDOMNode            react/no-find-dom-node
  13:5   error  Unexpected console statement       no-console
  13:17  error  Do not use findDOMNode            react/no-find-dom-node
```

Flow (https://flowtype.org, `npm name flow-bin`) — инструмент статической проверки типов, с помощью которого можно добавить специальный комментарий (`// @flow`) к сценариям и типам. Да! Типы в JavaScript! Радуйтесь, программисты, предпочитающие языки с сильной типизацией, такие как Java, Python и C. После добавления этих комментариев можно запустить Flow и проверить, будут ли обнаружены какие-либо проблемы. Эта программа тоже помогает предотвратить некоторые коварные ошибки:

```
// @flow
var bookName: string = 13
console.log(bookName) // Число - тип несовместим со строкой.
```

У Flow существует подробная документация: см. «Getting started with Flow» (https://flowtype.org/docs/getting-started.html) и «Flow for React» (https://flowtype.org/docs/react.html).

Atom или любой другой современный редактор кода можно настроить для интеграции ESLint и Flow, чтобы проблемы обнаруживались «на ходу».



Редактор кода Atom поддерживает Flow. На нижней панели выводится информация об обнаруженных проблемах, а в строках кода — соответствующие пометки

Код проекта email с ESLint v3.8.1 и Flow v0.33.0 находится в папке ch12/email-webpack-eslint-flow.

12.5. Оперативная замена модулей

Оперативная замена модулей (HMR, Hot Module Replacement) — одна из самых замечательных возможностей Webpack и React. Она позволяет написать код и быстрее протестировать его, обновляя браузер без потери состояния приложения.

Предположим, вы работаете над сложным одностраничным веб-приложением, и для перехода к текущей странице, над которой вы работаете, требуется около 12 щелчков. Если вы обновите сайт новым кодом, то для его выполнения необходимо щелкнуть на кнопке Reload/Refresh в браузере и повторить эти 12 щелчков. С другой стороны, при использовании HMR страницы не перезагружаются, а изменения отражаются прямо на странице.

Главным преимуществом HMR является ускорение итераций (написание кода, тестирование, написание кода, тестирование, и т. д.), потому что приложение будет сохранять состояние при внесении изменений. Некоторые разработчики считают HMR настолько замечательной возможностью, что если бы у React не было никакой другой функциональности, они бы все равно использовали React только ради HMR!

За техническими подробностями о том, как работает процесс HMR, обращайтесь к документации по адресу http://mng.bz/L9d5. В этом разделе рассмотрено практическое применение этой технологии к форме из примера с электронной почтой.

Процесс оперативного обновления кода состоит из нескольких шагов, показанных в упрощенной форме на рис. 12.2. Webpack HMR и сервер разработки используют WebSockets для отслеживания уведомлений об обновлениях от сервера. Если уведомления имеются, клиентская часть получает фрагменты (код JavaScript) и манифест обновления (JSON), который, по сути, представляет дельту изменений. Приложение клиентской части сохраняет свое состояние (например, данные в поле ввода или экранная позиция), но пользовательский интерфейс и код изменяются. Невероятно!

Чтобы увидеть HMR в действии, мы используем новый конфигурационный файл и `webpack-dev-server` (WDS). Вы можете использовать HMR со своим сервером, построенным с Express/Node; WDS не является обязательным компонентом, но предоставляется Webpack в отдельном модуле `webpack-dev-server`, поэтому я рассмотрю его здесь.

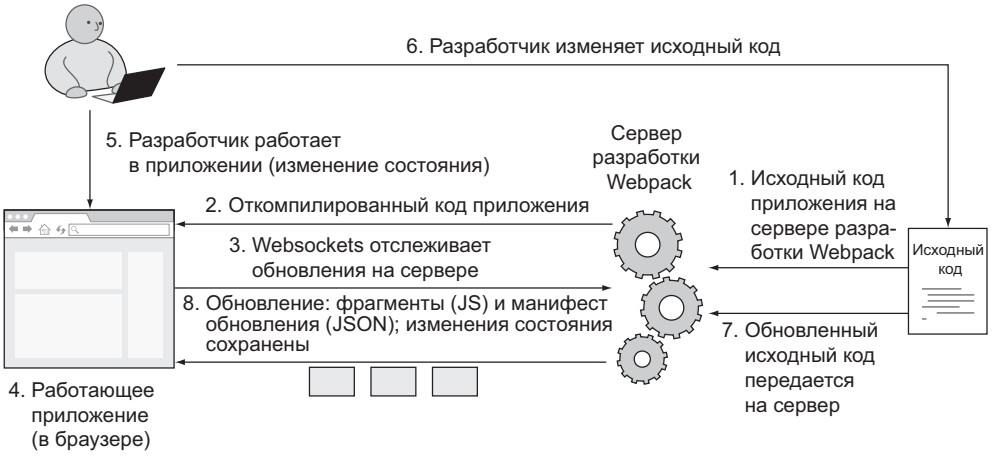


Рис. 12.2. Webpack отслеживает изменения в коде и отправляет сообщения об обновлениях выполняющемуся приложению в браузере

Когда все будет настроено, вы можете ввести адрес электронной почты в форму и внести несколько изменений в коде. Благодаря HMR вы увидите, что введенный адрес остался в форме, а изменения отражены в веб-приложении.

12.5.1. Настройка HMR

Для начала создайте копию webpack.config.js с именем webpack.dev.config.js:

```
$ cp webpack.config.js webpack.dev.config.js
```

Затем откройте только что созданный файл webpack.dev.config.js. В него нужно добавить несколько записей (новые точки входа, общедоступный путь, плагин HMR), а также задать флагу `dev-server` значение `true`. Окончательная версия файла представлена в листинге 12.4 (ch12/email-webpack/webpack.dev.config.js).

Листинг 12.4. webpack-dev-server и конфигурация HMR

```
const webpack = require('webpack')

module.exports = {
  entry: [
    'webpack-dev-server/client/?http://localhost:8080',
    './jsx/app.jsx'
  ],
  output: {
    publicPath: 'js/',
    path: __dirname + '/js/',
    filename: 'bundle.js'
  },
  devtool: '#sourcemap',
  module: {
    loaders: [
      { test: /\.css$/, loader: 'style-loader!css-loader' },
      {
        test: /\.jsx?$/,
        exclude: /(node_modules)/,
        loaders: ['react-hot-loader', 'babel-loader']
      }
    ]
  },
  devServer: {
    hot: true
  },
  plugins: [new webpack.HotModuleReplacementPlugin()]
}
```

Импортирует модуль Webpack

Включает WDS

Включает основное приложение

Задает путь к WDS, чтобы предоставить доступ к bundle.js (без записи на диск)

Включает react-hot-loader для автоматического включения HMR для всех файлов JSX

Включает режим HMR для WDS

Включает плагин HMR

Чтобы приказать WDS использовать новый конфигурационный файл, передайте ключ `--config`:

```
./node_modules/.bin/webpack-dev-server --config webpack.dev.config.js
```

Сохраните эту команду в package.json для удобства, если она еще не была сохранена ранее. Напомню, что `react-hot-loader` входит в число зависимостей. Этот модуль включает HMR для всех файлов JSX (которые, в свою очередь, преобразуются в JS).

Я предпочитаю включать HMR для всех файлов при помощи `react-hot-loader`. Но если вы хотите включить HMR только для части модулей, не используйте `react-hot-loader`; вместо этого следует добавить вызов `module.hot.accept()` в модули JSX/JS, которые вы хотите выбрать для HMR. Волшебство `module.hot` обеспечивает Webpack. В приложении рекомендуется проверить доступность `module.hot`:

```
if(module.hot) {
  module.hot.accept()
}
```

Многовато настроек! Однако существует другой способ использования и конфигурирования Webpack: вы можете использовать параметры командной строки и упаковать некоторые настройки в командах.

Если вы предпочитаете использовать командную строку — пожалуйста. Ваш конфигурационный файл получится более компактным, но сами команды будут больше. Например, файл webpack.dev-cli.config.js содержит меньше данных конфигурации:

```
module.exports = {
  entry: './jsx/app.jsx',
  output: {
    publicPath: 'js/',
    path: __dirname + '/js/',
    filename: 'bundle.js'
  },
  devtool: '#sourcemap',
  module: {
    loaders: [
      {
        test: /\.jsx?$/,
        exclude: /(node_modules)/,
        loaders: []
      }
    ]
  }
}
```

Но при этом он использует больше ключей CLI:

```
./node_modules/.bin/webpack-dev-server --inline --hot
➥ --module-bind 'css=style-loader!css-loader'
➥ --module-bind 'jsx=react-hot-loader!babel-loader'
➥ --config webpack.dev-cli.config.js
```

Ключи `--inline` и `--hot` активизируют WDS и режим HMR. Затем для передачи загрузчиков используется параметр `--module-bind` со следующим синтаксисом:

```
fileExtension=loader1!loader2!...
```

Проследите за тем, чтобы модуль `react-hot` предшествовал `babel`; в противном случае произойдет ошибка.

В том, что касается выбора между CLI и полным конфигурационным файлом, выбор за вами. Я считаю, что решение с CLI лучше подходит для более простых сборок. Чтобы не жалеть позднее, когда вы обнаружите, что при вводе чудовищно длинной команды была совершена опечатка, сохраните команду как сценарий npm в package.json. Нет, пакетные файлы/сценарии оболочки/make-сценарии уже не в моде. Используйте сценарии npm, как все современные люди! (Дисклеймер: это шутка. Я не сторонник разработки, управляемой модой.)

СЦЕНАРИИ NPM

Сценарии npm обладают рядом преимуществ и широко используются в проектах Node и React. Они стали фактическим стандартом, и вы обычно сталкиваетесь с ними при первом знакомстве с проектом. Начиная работать над новым проектом или библиотекой, я после readme.md первым делом обращаюсь к сценариям npm — а иногда и вместо файла readme.md, который может содержать устаревшую информацию.

Сценарии npm предоставляют гибкие возможности сохранения важнейших сценариев для тестирования, сборки, заполнения данными и выполнения в ходе разработки или в других средах. Иначе говоря, любая работа, выполняемая в CLI и связанная с приложением, но не принадлежащая самому приложению, может быть сохранена в сценариях `npm`. Кроме того, сценарии npm решают задачу документирования, показывая другим, как работает сборка и тестирование. Из сценариев npm также можно вызывать другие сценарии npm, что приводит к еще более значительному упрощению проекта. Следующий пример включает разные версии сборок:

```
"scripts": {
  "test": "echo \"Error: no test specified\" && exit 1",
  "build": "./node_modules/.bin/babel -w",
  "build:method": " npm run build -- method/jsx/script.jsx -o
  ➥ method/js/script.js",
  "build:hello-js-world-jsx": "npm run build --
  ➥ hello-js-world-jsx/jsx/script.jsx -o
  ➥ hello-js-world-jsx/js/script.js",
  "build:hello-world-jsx": "npm run build --
  ➥ hello-world-jsx/jsx/script.jsx -o
  ➥ hello-world-jsx/js/script.js",
  "build:hello-world-class-jsx": "npm run build --
  ➥ hello-world-class-jsx/jsx/script.jsx -o
  ➥ hello-world-class-jsx/js/script.js"
},
```

Сценарии npm также поддерживают pre- и post-перехватчики, которые делают их еще более универсальными. В общем случае перехватчик (hook) представляет собой паттерн, в котором присутствует некоторый код, выполняемый при возникновении другого события. Например, вы можете создать задачу `learn-react` с двумя задачами, имеющими `pre-` и `post`-перехватчики: `prelearn-react` и `postlearn-react`. Как вы, возможно, уже догадались, pre-перехватчик выполняется до learn-react, а post-перехватчик выполняется после `learn-react`. Например, следующие сценарии `bash`:

```
"scripts": {
    "prelearn-react": "echo \"Purchasing React Quickly\"",
    "learn-react": "echo \"Reading React Quickly\" ",
    "postlearn-react": "echo \"Creating my own React app\""
  },
```

выдают следующие результаты в соответствии с порядком `pre/post`:

```
...
Purchasing React Quickly
...
Reading React Quickly
...
Creating my own React app
```

С `pre-` и `post`-перехватчиками `npm` может легко заменить некоторые шаги сборки, выполняемые Webpack, Gulp и Grunt.

За дополнительной информацией о npm, включая параметры и аргументы, обращайтесь к документации по адресу https://docs.npmjs.com/misc/scripts и статье Кейт Сиркл (Keith Cirkel) «How to Use npm as a Build Tool» (www.keithcirkel.co.uk/how-to-use-npm-as-a-build-tool). Любая функциональность, отсутствующая в сценариях npm, может быть реализована с нуля в виде сценария Node. К преимуществам этого варианта можно отнести меньшее количество зависимостей от плагинов вашего проекта.

12.5.2. Оперативная замена модулей в действии

Запустите WDS командой npm run wds или npm run wds-cli. Затем откройте адрес http://localhost:8080 и откройте консоль DevTools. Вы увидите сообщения от HMR и WDS:

```
[HMR] Waiting for update signal from WDS...
[WDS] Hot Module Replacement enabled.
```

Введите текст в поле для адреса электронной почты или комментария, а затем измените content.jsx. Вы можете изменить что-нибудь в `render()` — например, поменять текст формы с `Email` на `Your Email`:

```
Your Email: <input ref="emailAddress" className="form-control" type="text"
➥ placeholder="hi@azat.co"/>
```

В консоли появляются сообщения:

```
[WDS] App updated. Recompiling...
...
[HMR] App is up to date.
```

Ваши изменения отражаются на веб-странице (см. рис. 12.3) вместе с текстом, который был введен ранее. Отлично! Вам больше не нужно тратить время на ввод

тестовых данных или навигацию по нескольким вложенным интерфейсам! Вы сможете тратить время на содержательную работу, вместо того чтобы вводить текст и щелкать мышью в разных местах приложения клиентской части. HMR ускоряет разработку!

ПРИМЕЧАНИЕ Механизм HMR не идеален. В некоторых ситуациях он не обнаруживает изменения или приводит к фатальному сбою. Когда это происходит, WDS перезагружает страницу (живая перезагрузка). Это поведение определяется параметром `webpack/hot/dev-server`; другой вариант — ручная перезагрузка с использованием `webpack/hot/only-dev-server`.

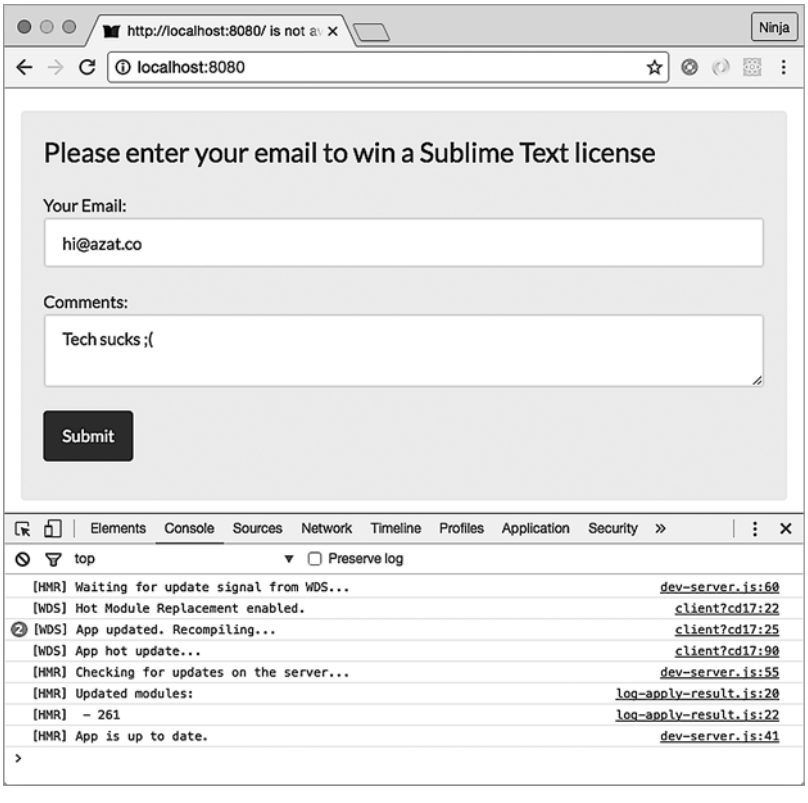


Рис. 12.3. HMR обновляет представление, заменяя «Email» на «Your Email», без стирания данных в полях

Webpack — удобный инструмент, используемый с React для оптимизации и усовершенствования упаковки. Благодаря WDS и HMR он прекрасно подходит для оптимизации кода, изображений, стилей и других активов не только при развертывании, но и при разработке.

12.6. Вопросы

1. Какая команда будет использоваться для сценария npm (`"dev": "./node_modules/.bin/webpack-dev-server --config webpack.dev.config.js"`): `npm dev`, `npm run dev`, `NODE_ENV=dev npm run` или `npm run development`?
2. HMR — всего лишь термин React, обозначающий «живую» перезагрузку. Да или нет?
3. WDS записывает откомпилированные файлы на диск, как и команда `webpack`. Да или нет?
4. Файл webpack.config.js должен содержать действительный формат JSON, как и package.json. Да или нет?
5. Какие загрузчики необходимо использовать для импортирования и последующего внедрения CSS в веб-страницу с использованием Webpack?

12.7. Итоги

- Для работы оперативной замены модулей необходимо включить `webpack-dev-server` и `react-hot-loader` в конфигурационный файл или вызов `module.hot.accept()` в соответствующие файлы.
- Также можно использовать `require()` для загрузки CSS с `style-loader` и `css-loader`.
- `devtool: '#sourcemap'` обеспечивает поддержку нумерации строк в откомпилированном коде.
- Параметр WDS `publicPath` сообщает WDS, где должен размещаться пакет.

12.8. Ответы

1. `npm run dev`. Только сценарии npm `start` и `test` могут выполняться без `run`. Все остальные сценарии должны использовать синтаксис `npm run NAME`.
2. Нет. HMR может заменить «живую» перезагрузку и вернуться к ней при отказе HMR; однако HMR обладает большими возможностями и предоставляет больше возможностей, таких как обновление отдельных частей приложения с сохранением состояния.
3. Нет. WDS только предоставляет файлы, не записывая их на диск.
4. Нет. `webpack.config.js` — конфигурационный файл Webpack по умолчанию. Это должен быть файл NodeJS/JavaScript с модулем CommonJS/NodeJS, экспортирующий объектный литерал для конфигураций (объект может заключаться в двойные кавычки по аналогии с JSON).
5. Загрузчик `style` обеспечивает импортирование, а загрузчик CSS — внедрение.

13 Маршрутизация в React

Посмотрите вступительный видеоролик к этой главе, отсканировав QR-код или перейдя по ссылке http://reactquickly.co/videos/ch13.

ЭТА ГЛАВА ОХВАТЫВАЕТ СЛЕДУЮЩИЕ ТЕМЫ:

- Реализация маршрутизации с нуля.
- Функциональность React Router.
- Маршрутизация с использованием Backbone.

В прошлом во многих одностраничных приложениях URL-адрес редко изменялся в процессе работы с приложением. Благодаря рендерингу в браузере обращаться к серверу не было необходимости! Изменился только контент в части страницы. Такой подход обладает рядом недостатков:

- Обновление браузера возвращало пользователя к исходной форме страницы.
- Щелчок на кнопке Назад в браузере возвращал пользователя к другому сайту, потому что в истории браузера регистрировался только один URL-адрес сайта. В истории не сохранялись изменения URL, отражающие навигацию по контенту.
- Пользователь не мог отправить друзьям ссылку на конкретную страницу сайта.
- Поисковые системы не могли проиндексировать страницу из-за отсутствия разных URL для индексирования.

К счастью, в наши дни появился механизм *браузерной маршрутизации URL*. Маршрутизация URL позволяет настроить приложение для получения URL-адресов запросов, не соответствующих физическим файлам. Вместо этого вы определяете URL-адреса, семантически содержательные с точки зрения пользователя; с ними становится возможной поисковая оптимизация (SEO), и они могут отражать со-

стояние приложения. Например, URL-адрес страницы для вывода информации о продукте может выглядеть так:

```
https://www.manning.com/books/react-quickly
```

Во внутренней реализации он отображается на страницу, на которой выводится продукт с идентификатором `react-quickly`. При просмотре разных продуктов URL может изменяться, и как браузер, так и поисковые системы смогут взаимодействовать со страницами продуктов так, как вы ожидаете. Если вы хотите избежать полной перезагрузки страниц, включите в URL символ `#`, как это делают многие известные сайты:

```
https://mail.google.com/mail/u/0/#inbox
https://en.todoist.com/app?v=816#agenda%2Foverdue%2C%20today
https://calendar.google.com/calendar/render?tab=mc#main_7
```

Маршрутизация URL обязательна для удобного, хорошо спроектированного веб-приложения. Без конкретных URL-адресов пользователи не смогут сохранять ссылки или обмениваться ими без потери состояния приложения, будь то одностраничное приложение (SPA) или традиционное веб-приложение с серверным рендерингом.

В этой главе мы построим простой веб-сайт на базе React и рассмотрим пару вариантов реализации маршрутизации на этом сайте. Библиотека React Router будет представлена позднее; сначала мы построим простейшую маршрутизацию «с нуля».

> **ПРИМЕЧАНИЕ** Исходный код примеров этой главы доступен по адресам www.manning.com/books/react-quickly и https://github.com/azat-co/react-quickly/tree/master/ch13 (в папке ch13 репозитория GitHub https://github.com/azat-co/react-quickly). Также некоторые демонстрационные примеры доступны по адресу http://reactquickly.co/demos.

13.1. Реализация маршрутизации с нуля

Существуют библиотеки, реализующие маршрутизацию для React, но мы начнем с самостоятельной реализации — чтобы вы увидели, как это просто. Проект также поможет вам понять, как работают другие системы маршрутизации.

Конечная цель проекта — создать три страницы, изменяющиеся вместе с URL в ходе навигации. Для простоты будут использоваться URL с символом `#`; другие URL требуют специальной настройки сервера. Будут созданы следующие страницы:

- `Home` — `/` (пустой путь в URL)
- `Accounts` — `/#accounts`
- `Profile` — `/#profile`

На рис. 13.1 представлена схема навигации от домашней страницы к странице `Profile`.

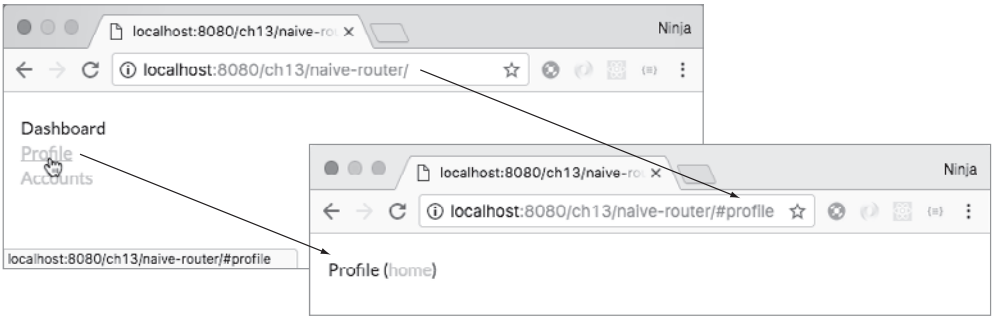


Рис. 13.1. Навигация от домашней страницы к странице Profile и изменение URL-адреса щелчком на ссылке

Чтобы реализовать этот проект, который продемонстрирует использование маршрутизатора URL, мы создадим компонент-маршрутизатор (router.jsx), мэппинг и страницу HTML. Компонент-маршрутизатор будет получать информацию из URL и обновлять веб-страницу соответствующим образом. Реализация проекта делится на следующие этапы.

1. Создание *мэппинга,* то есть соответствия между введенным URL-адресом и ресурсами (элементами React или компонентами). Мэппинг привязан к конкретному проекту, и для каждого нового проекта придется создавать новый.
2. Написание библиотеки маршрутизации с нуля. Она берет запрашиваемый URL-адрес и проверяет его по мэппингу (этап 1). В router.jsx библиотека маршрутизации будет представлять собой один компонент `Router`, который может повторно использоваться в различных проектах.
3. Написание приложения-примера, в котором будет использоваться компонент `Router` с этапа 2 и мэппинг с этапа 1.

Для создания элементов React для разметки будет использоваться JSX. Разумеется, `Router` не обязательно оформлять как компонент React; это может быть обычная функция или класс. Тем не менее использование компонента React поможет закрепить концепции, изложенные в книге, такие как жизненные циклы событий, механизм рендеринга React и работа с моделью DOM. Кроме того, ваша реализация будет ближе к реализации маршрутизации в React Router, а это поможет вам лучше понять маршрутизацию React Router, когда мы будем рассматривать ее позднее.

13.1.1. Создание проекта

Проект (который можно было бы назвать *простым,* или *наивным*, маршрутизатором) имеет следующую структуру:

```
/naive-router
  /css
    bootstrap.css
    main.css
  /js
    bundle.js
  /jsx
    app.jsx
    router.jsx
  /node_modules
  index.html
  package.json
  webpack.config.js
```

Начнем с установки зависимостей. Я помещу их в package.json; вы можете скопировать зависимости вместе с конфигурацией `babel` и `scripts` и выполнить команду `npm install` (ch13/naive-router/package.json).

Листинг 13.1. Подготовка среды разработки

```
{
  "name": "naive-router",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "build": "./node_modules/.bin/webpack -w"
  },
  "author": "Azat Mardan",
  "license": "MIT",
  "babel": {
    "presets": [
      "react"
    ]
  },
  "devDependencies": {
    "babel-core": "6.18.2",
    "babel-loader": "6.2.4",
    "babel-preset-react": "6.5.0",
    "webpack": "2.4.1"
    "react": "15.5.4",
    "react-dom": "15.5.4"
  },
  "dependencies": {
  }
}
```

Сохраняет сценарий сборки Webpack как сценарий npm

Сообщает Babel, какие конфигурации следует использовать (React для JSX в данном случае; конфигурация ES6+ не обязательна)

Устанавливает Webpack 2.4.1 локально (рекомендуется)

Но это еще не все. Webpack нужен собственный конфигурационный файл webpack.config.js (см. главу 9). Здесь необходимо настроить *источник* (`entry`) и предполагаемый *приемник* (`output`), а также указать загрузчик.

Листинг 13.2. webpack.config.js

```
module.exports = {
  entry: './jsx/app.jsx',
  output: {
    path: __dirname + '/js/',
    filename: 'bundle.js'
  },
  module: {
    loaders: [
      {
        test: /\.jsx?$/,
        exclude: /(node_modules)/,
        loader: 'babel-loader'
      }
    ]
  }
}
```

Определяет начальный файл для сборки (как правило, это основной файл, который загружает другие файлы)

Определяет путь для упакованных файлов

Определяет имя пакетного файла, которое будет использоваться в index.html

Задает загрузчик, который выполнит преобразование JSX (и ES6+ при необходимости)

13.1.2. Создание мэппинга маршрутов в файле app.jsx

Сначала следует создать мэппинг с объектом `mapping`, ключами которого являются фрагменты URL, а значениями — контроллер конкретных страниц. Мэппинг получает значение и связывает его с другим значением. В данном случае ключ (фрагмент URL) отображается на JSX. При желании можно создать отдельный файл для каждой страницы, но мы объединим все отображения в файле app.jsx.

Листинг 13.3. Мэппинг маршрутов (app.jsx)

```
const React = require('react')
const ReactDOM = require ('react-dom')
const Router = require('./router.jsx')

const mapping = {
  '#profile': <div >Profile (<a href="#">home</a>)</div>,
  '#accounts': <div >Accounts (<a href="#">home</a>)</div>,
  '*': <div>Dashboard<br/>
    <a href="#profile">Profile</a>
    <br/>
    <a href="#accounts">Accounts</a>
    </div>
}

ReactDOM.render(
  <Router mapping = {mapping}/>,
  document.getElementById('content')
)
```

Использует CommonJS require() для импортирования модулей с упаковкой Webpack

Использует объект mapping, связывающий маршруты с отдельными страницами

Передает мэппинг Router

Реализация компонента `Router` содержится в файле router.jsx.

13.1.3. Создание компонента Router в router.jsx

По сути, компонент `Router` должен получить информацию из URL (`#profile`) и распределить ее на JSX по данным свойства `mapping`. Для обращения к URL можно воспользоваться свойством `window.location.hash` API браузера:

```
const React = require('react')
module.exports = class Router extends React.Component {
  constructor(props) {
    super(props)
    this.state = {hash: window.location.hash}
    this.updateHash = this.updateHash.bind(this)
  }
  render() {
    ...
  }
}
```

Далее необходимо прослушивать любые изменения в URL при помощи `hashchange`. Если вы не реализуете прослушивание новых URL-адресов, маршрутизатор сработает только один раз: при перезагрузке всей страницы и создании элемента `Router`. Лучшие места для присоединения и отсоединения слушателей `hashchange` — обработчики событий жизненного цикла `componentDidMount()` и `componentWillUnmount()`:

```
updateHash(event) {
  this.setState({hash: window.location.hash})
}
componentDidMount() {
  window.addEventListener('hashchange', this.updateHash, false)
}
componentWillUnmount() {
  window.removeEventListener('hashchange', this.updateHash, false)
}
```

COMPONENTDIDMOUNT() И COMPONENTWILLUNMOUNT()

События жизненного цикла рассматриваются в главе 5, но я напомню в двух словах. Событие `refresher.componentDidMount()` срабатывает при подключении элемента и появляется в реальной модели DOM (можно сказать, что элементу соответствует узел реальной модели DOM). По этой причине этот обработчик является самым безопасным местом для присоединения событий, интегрируемых с другими объектами DOM, и для вызовов AJAX/XHR (здесь не используются).

С другой стороны, `componentWillUnmount()` лучше всего подходит для отсоединения слушателей событий; элемент отключается, и вам нужно отсоединить все, что было создано за пределами этого элемента (например, слушатель события для window). Оставлять слушатели событий без элементов, которые создали и использовали их, не рекомендуется: это приводит к таким проблемам, как утечка памяти.

В `render()` можно воспользоваться конструкцией `if/else`, чтобы увидеть совпадение с текущим URL-адресом (`this.state.hash`) с ключами/атрибутами/свойствами в свойстве `mapping`. Если соответствие будет обнаружено, вы снова обращаетесь к `mapping` для получения контента конкретной страницы (JSX), если нет — происходит возврат к разделу * для всех остальных URL, включая пустое значение (домашняя страница). Полный код приведен в листинге 13.4 (ch13/naive-router/jsx/router.jsx).

Листинг 13.4. Реализация маршрутизатора URL

```
const React = require('react')
module.exports = class Router extends React.Component {
  constructor(props) {
    super(props)
    this.state = {hash: window.location.hash}
    this.updateHash = this.updateHash.bind(this)
  }
  updateHash(event) {
    this.setState({hash: window.location.hash})
  }
  componentDidMount() {
    window.addEventListener('hashchange', this.updateHash, false)
  }
  componentWillUnmount() {
    window.removeEventListener('hashchange', this.updateHash, false)
  }
  render() {
    if (this.props.mapping[this.state.hash])
      return this.props.mapping[this.state.hash]
    else
      return this.props.mapping['*']
  }
}
```

Присваивает исходное значение идентификатора фрагмента

Предоставляет новые значения идентификатора фрагмента

Выполняет рендер контента, соответствующего идентификатору фрагмента

Наконец, в файле index.html включается файл CSS и файл bundle.js, который будет сгенерирован Webpack при выполнении команды `npm run build` (которая, в свою очередь, выполнит команду `./node_modules/.bin/webpack -w`):

```
<!DOCTYPE html>
<html>

  <head>
    <link href="css/bootstrap.css" type="text/css" rel="stylesheet"/>
    <link href="css/main.css" type="text/css" rel="stylesheet"/>
  </head>

  <body>
    <div id="content" class="container"></div>
    <script src="js/bundle.js"></script>
  </body>

</html>
```

Запустите сборщик, чтобы получить файл bundle.js, и откройте веб-страницу в браузере. Щелчки на ссылках изменяют как URL, так и контент страницы (см. рис. 13.1).

Как видите, построить собственный маршрутизатор с React, в общем-то, несложно; вы можете использовать методы жизненного цикла для отслеживания изменений в идентификаторе фрагмента и выводе соответствующего контента. Но хотя это решение остается теоретически возможным, ситуация быстро усложняется, если вам нужны вложенные маршруты, разбор маршрутов (извлечение параметров URL) или «красивые» URL-адреса без #. Конечно, можно воспользоваться маршрутизатором из Backbone или другого MVC-образного фреймворка, но существует решение, спроектированное специально для React (подсказка: в нем используется JSX).

13.2. React Router

React прекрасно подходит для построения пользовательских интерфейсов. Если я вас еще не убедил, вернитесь и перечитайте предыдущие главы! React также может использоваться для реализации простой маршрутизации URL с нуля, как было показано для router.jsx.

Но для более сложных одностраничных приложений (SPA) требуется более широкая функциональность. Например, передача параметра в URL — стандартный способ передачи отдельных элементов вместо списка: например, `/posts/57b0ed12fa81dea5362e5e98`, где `57b0ed12fa81dea5362e5e98` — уникальный идентификатор сообщения. Для извлечения параметра из URL можно воспользоваться регулярным выражением; но рано или поздно, когда сложность вашего приложения возрастет, вам, возможно, придется заново изобретать уже существующие реализации маршрутизации URL.

СЕМАНТИЧЕСКИЕ URL-АДРЕСА

Семантические URL-адреса (https://ru.wikipedia.org/wiki/Семантический_URL) улучшают удобство использования и доступность веб-сайта или веб-приложения за счет отделения внутренней реализации от пользовательского интерфейса. В несемантическом решении могут использоваться строки запросов и/или имена файлов сценариев. С другой стороны, при семантическом подходе пути используются только таким способом, который помогает пользователям интерпретировать структуру и работать с URL. Приведу несколько примеров.

Несемантические URL (неплохо)	Семантические URL (лучше)
http://webapplog.com/show?post=es6	http://webapplog.com/es6
https://www.manning.com/books/react-quickly?a_aid=a&a_bid=5064a2d3	https://www.manning.com/books/react-quickly/a/5064a2d3
http://en.wikipedia.org/w/index.php?title=Semantic_URL	https://en.wikipedia.org/wiki/Semantic_URL

Многие фреймворки, такие как Ember, Backbone и Angular, содержат встроенные механизмы маршрутизации. Что касается маршрутизации в React, существует React Router (`react-router`; https://github.com/reactjs/react-router) — готовое, полноценное решение. В разделе 13.4 будет рассмотрена реализация для Backbone; там показано, как хорошо React сочетается с этим фреймворком в стиле MVC, который используется многими разработчиками для построения SPA-приложений. А пока сосредоточимся на React Router.

React Router не входит в официальную базовую библиотеку React. Эта библиотека была разработана сообществом, но она достаточно популярна и проверена временем, чтобы использоваться в трети проектов React[1]. Большинство React-разработчиков, с которыми я общался, выбирает ее по умолчанию.

Синтаксис React Router использует JSX, и это дополнительный плюс, поскольку разработчик может создавать более удобочитаемые иерархические определения, чем с объектом `mapping` (как в предыдущем проекте). Как и наша наивная реализация, библиотека React Router содержит компонент React `Router` (и он в каком-то смысле вдохновил мою реализацию!). Основные шаги, которые необходимо выполнить для его использования:

1. Создайте мэппинг, преобразующий URL в компоненты React (которые преобразуются в разметку на веб-странице). В React Router эта задача решается передачей свойств `path` и `component`, а также внешнего объекта `Route`. Мэппинг определяется в JSX объявлением и вложением компонентов Route. Эта часть должна быть реализована для каждого нового проекта.
2. Используйте компоненты React Router `Router` и `Route`, которые творят волшебство с изменением представлений в соответствии с изменениями в URL. Разумеется, эту часть реализовать вам не придется, но библиотеку нужно установить.
3. Выполните рендер компонента `Router` на веб-странице, подключая его к `ReactDOM.render()` как обычный элемент React. Не стоит и говорить, что эта часть должна быть реализована для каждого нового проекта.

JSX используется для создания объекта `Route` для каждой страницы, а также для вложения его в другой объект `Route` или `Router`. Объект `Router` находится в функции `ReactDOM.render()`, как и любой другой элемент React:

```
ReactDOM.render((
  <Router ...>
    <Route ...>
      <Route ../>
      ...
    </Route>
    <Route .../>
  </Router>
), document.getElementById('content'))
```

[1] React.js Conf 2015, «React Router Increases Your Productivity», https://youtube.com/watch?v=XZfvW1a8Xac.

Каждый объект `Route` содержит как минимум два свойства: `path` (шаблон URL, совпадение которого активизирует маршрут) и `component` (выборка и рендеринг необходимого компонента). Для `Route` также можно определить другие свойства, например обработчики событий и данные. Они будут доступны в свойстве `props.route` этого компонента `Route`. Так происходит передача данных компонентам маршрутов.

Для демонстрации рассмотрим пример SPA-приложения с маршрутизацией по нескольким страницам: About, Posts (некое подобие блога), Post, Contact Us и Login. Эти страницы имеют разные пути и рендерятся разными компонентами:

- About — `/about`
- Posts –`/posts`
- Post — `/post`
- Contact — `/contact`

Страницы About, Posts, Post и Contact Us имеют одинаковый макет (компонент `Content`) и осуществляют рендер внутри него. Начальная (не финальная!) версия кода с использованием React Router выглядит так:

```
<Router>
  <Route path="/" component={Content} >
    <Route path="/about" component={About} />
      <Route path="/about/company" .../>
      <Route path="/about/author" .../>
    <Route path="/posts" component={Posts} />
    <Route path="/posts/:id" component={Post}/>
    <Route path="/contact" component={Contact} />
  </Route>
</Router>
```

Интересно, что маршруты могут вкладываться для повторного использования макетов из родителей, а их URL-адреса могут быть независимыми от вложения. Например, возможно иметь вложенный компонент `About` с URL-адресом `/about`, хотя маршрут «родительского» макета `Content` использует `/app`. Компонент `About` также будет содержать макет `Content` (реализованный посредством `this.props.children` в `Content`):

```
<Router>
  <Route path="/app" component={Content} >
    <Route path="/about" component={About} />
    ...
```

Другими словами, `About` не требуется вложенный URL-адрес `/app/about`, если только вы не захотите, чтобы это было именно так. Тем самым достигается большая гибкость в отношении путей и макетов.

Для навигации будет реализовано меню, изображенное на рис. 13.2. Меню и заголовок будут отрендерены из `Content` и повторно использованы на страницах About, Posts, Post и Contact Us. На иллюстрации стоит обратить внимание на ряд моментов:

рендерится страница About, кнопка в меню активна, URL отражает тот факт, что текущей является страница About (секция `/#/about`), а текст `Node.University` отражает текущее содержимое компонента `About` (об этом позднее).

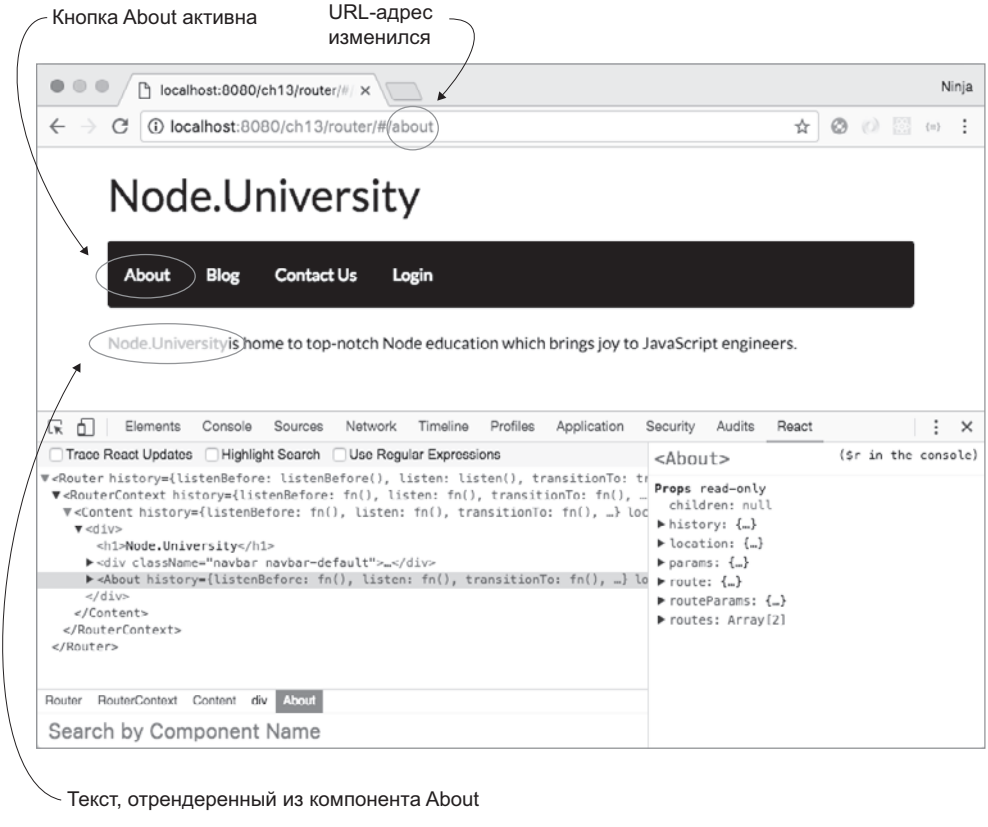


Рис. 13.2. При переходе к странице About выводится текст About в компоненте Component, изменяется URL-адрес, а кнопка становится активной

13.2.1. Стиль JSX в React Router

Как упоминалось ранее, мы воспользуемся JSX для создания элемента `Router` и элементов `Route`, вложенных в него (и друг в друга). Каждый элемент (`Router` или `Route`) содержит как минимум два свойства — `path` и `component`, которые сообщают маршрутизатору путь URL и класс компонента React для создания и рендера. Возможно наличие дополнительных нестандартных свойств/атрибутов для передачи данных; мы используем эту возможность для передачи массива сообщений.

Итак, пора применить новые знания на практике: импортируйте объекты React Router и используйте их в `ReactDOM.render()` для определения поведения маршру-

тизации (ch13/router/jsx/app.jsx). Кроме About, Posts, Post и Contact Us, будет создана страница входа Login.

Листинг 13.5. Определение Router

```
const ReactRouter = require('react-router')
let { Router,
  Route,
  Link
} = ReactRouter

ReactDOM.render((
  <Router history={hashHistory}>
    <Route path="/" component={Content} >
      <Route path="/about" component={About} />
      <Route path="/posts" component={Posts} posts={posts}/>
      <Route path="/posts/:id" component={Post} posts={posts}/>
      <Route path="/contact" component={Contact} />
    </Route>
    <Route path="/login" component={Login}/>
  </Router>
), document.getElementById('content'))
```

Последний маршрут `Login` (`/login` — рис. 13.3) существует вне маршрута `Content` и не отображает меню (которое находится в `Content`). Все, чему не нужен общий интерфейс, предоставляемый `Content`, можно вывести за пределы маршрута `Content`. Это поведение определяется иерархией вложенных маршрутов.

Компонент `Post` рендерит информацию сообщения в блоге на основании описательной части (часть URL — можно считать, что это идентификатор сообщения), которую он читает из URL (например, `/posts/http2`) с использованием переменной `props.params.id`. Включая в путь специальный синтаксис с двоеточием, вы приказываете маршрутизатору разобрать это значение и использовать для заполнения `props.params`.

`Router` передается методу `ReactDOM.render()`. Обратите внимание на передачу `history` компоненту `Router`. Начиная с версии 2 React Router, необходимо передавать реализацию истории. Возможны два варианта: использование истории из React Router и использование автономной реализации истории.

13.2.2. История идентификатора фрагмента

История идентификатора фрагмента[1] зависит от символа `#`, который позволяет осуществлять навигацию по странице без ее перезагрузки (например, `router/#/posts/http2`). `#` используется во многих SPA-приложениях, потому что они должны

[1] Hash history — хеш url — это всё, что идет после символа #. В данном контексте хеш — это история. — *Примеч. ред.*

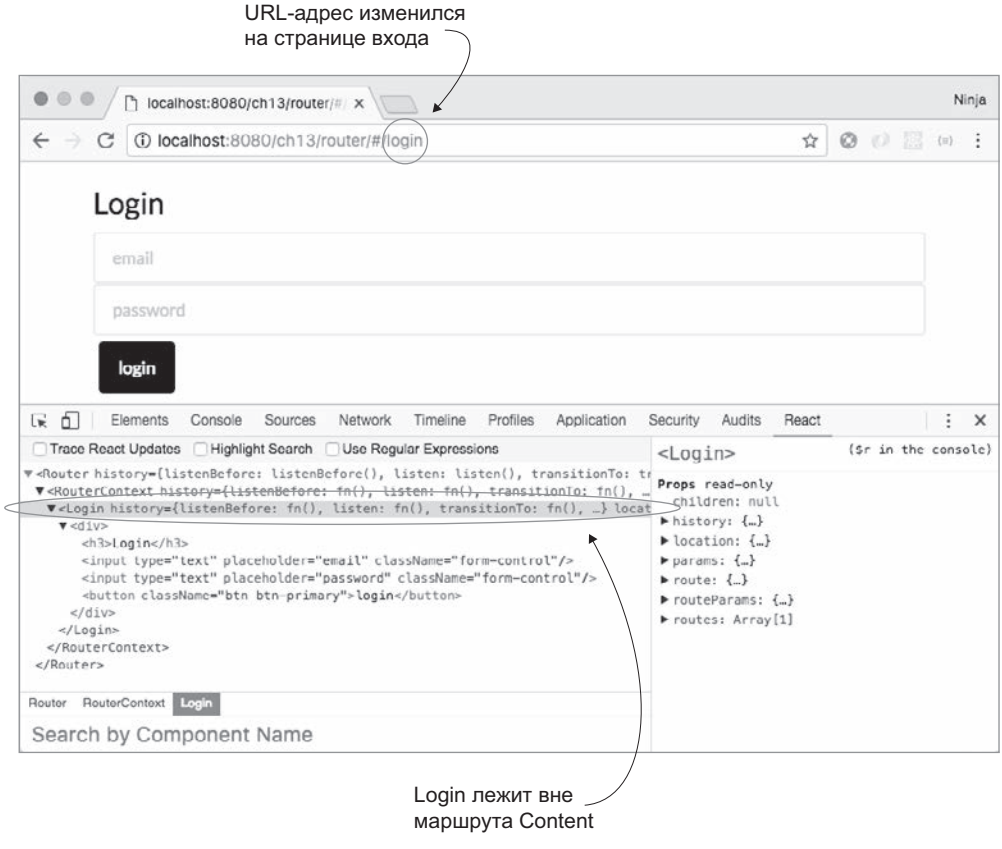


Рис. 13.3. Страница Login (/#/login) не использует общий макет (Content), включающий меню. Присутствует только элемент Login

отражать изменения в контексте без полного обновления (запроса к серверу). Мы тоже использовали его при реализации маршрутизатора с нуля.

> **ПРИМЕЧАНИЕ** Правильный термин для части URL-адреса, следующей за символом # — идентификатор фрагмента (https://en.wikipedia.org/wiki/Fragment_identifier).

В данном примере также используется идентификатор фрагментов, который предоставляется библиотекой `history` (http://npmjs.org/history). Нужно импортировать библиотеку, инициализировать ее и передать React Router.

При инициализации истории необходимо присвоить `queryKey` значение `false`, потому что нужно запретить строку запроса (например, `?_k=vl8reh`), присутствующую по умолчанию для поддержки старых браузеров и передачи состояний при навигации:

```
const ReactRouter = require('react-router')
const History = require('history')
let hashHistory = ReactRouter.useRouterHistory(History.createHashHistory)({
  queryKey: false
})
<Router history={hashHistory}/>
```

Чтобы использовать встроенную историю идентификатора фрагмента, импортируйте ее из React Router:

```
const { hashHistory } = require('react-router')
<Router history={hashHistory} />
```

При желании с React Router можно воспользоваться другой реализацией истории. Старые браузеры поддерживают историю идентификатора фрагмента, но это означает, что вы будете видеть символ `#`. Если вам нужны URL-адреса без `#`, это тоже возможно. Достаточно переключиться на историю браузера и реализовать некоторые изменения на сервере — простые, если вы используете Node в качестве серверной части HTTP. Чтобы не усложнять проект, мы воспользуемся историей идентификатора фрагмента, но вскоре доберемся и до истории браузера.

13.2.3. История браузера

Альтернативой для истории идентификатора фрагмента служит история браузера HTML5 `pushState`. Например, URL-адрес в истории браузера может иметь вид `router/posts/http2` вместо `router/#/posts/http2`. URL-адреса из истории браузера также называются *реальными* URL-адресами.

История браузера использует обычные, нефрагментированные URL-адреса, так что каждый запрос инициирует обращение к серверу. Вот почему этот способ требует определенной настройки на стороне сервера, которая здесь не рассматривается. Обычно SPA-приложения должны использовать фрагментированные URL-адреса, особенно если вы намерены поддерживать более старые браузеры, потому что история браузера требует более сложной реализации.

История браузера используется примерно так же, как история идентификатора фрагмента. Импортируйте модуль, подключите его и, наконец, настройте сервер для предоставления того же файла (не файла из маршрутизации SPA).

Браузерные реализации берутся из специального автономного пакета (как, например, `history`) или из реализации из React Router (`ReactRouter.browserHistory`). После этого импортируйте библиотеку истории браузера и примените ее к `Router`:

```
const { browserHistory } = require('react-router')
<Router history={browserHistory} />
```

Далее необходимо изменить сервер, чтобы он отвечал одним файлом независимо от URL. Этот пример — всего лишь одна из возможных реализаций; она использует Node.js и Express:

```
const express = require('express')
const path = require('path')
const port = process.env.PORT || 8080
const app = express()

app.use(express.static(__dirname + '/public'))

app.get('*', function (request, response){
  response.sendFile(path.resolve(__dirname, 'public', 'index.html'))
})

app.listen(port)
console.log("server started on port"+port)
```

Причина, по которой потребуется настройка поведения сервера HTTP, заключается в том, что при переключении на реальные URL-адреса без знака # они начнут попадать к серверу HTTP. Сервер должен поставлять один и тот же код SPA JavaScript на каждый запрос. Например, запросы к `/posts/57b0ed12fa81dea5362e5e98` и `/about` должны разрешаться в запросы index.html, а не posts/57b0ed12fa81dea5362e5e98.html или about.html (что, вероятно, приведет к ошибке *404: Not Found*).

Так как история идентификатора фрагмента является предпочтительным способом реализации маршрутизации URL, там, где требуется поддержка старых браузеров, а также для упрощения примера я буду использовать в этой главе историю идентификатора фрагмента.

13.2.4. Настройка React Router для разработки с Webpack

При работе с React Router необходимо использовать и импортировать определенные библиотеки, а также обеспечить компиляцию JSX. Рассмотрим настройку для React Router с использованием Webpack, который выполнит все эти действия.

В листинге 13.6 приведен раздел `devDependencies` из файла package.json (ch13/router/package.json). Большая часть его содержимого вам уже знакома. К числу новых пакетов относятся `history` и `react-router`. Как обычно, проследите за тем, чтобы использовались именно эти версии; в противном случае выполнение кода не гарантировано.

Листинг 13.6. Зависимости для использования Webpack v1, React Router v2.6, React v15.2 и JSX

```
{
  ...
  "devDependencies": {
    "babel-core": "6.11.4",
    "babel-loader": "6.2.4",
    "babel-preset-react": "6.5.0",
    "history": "2.1.2",
    "react": "15.2.1",
```

```
    "react-dom": "15.2.1",
    "react-router": "2.6.0",
    "webpack": "1.12.9"
  }
}
```

Кроме `devDependencies`, файл package.json должен содержать конфигурацию `babel`. Я также рекомендую добавить сценарии npm:

```
{
  ...
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "build": "./node_modules/.bin/webpack -w",
    "i": "rm -rf ./node_modules && npm cache clean && npm install"
  },
  "babel": {
    "presets": [
      "react"
    ]
  },
  ...
}
```

Так как код JSX будет преобразован в `React.createClass()`, вы должны импортировать и определить `React` в файлах, использующих JSX, даже если они не используют `React`. Скажем, в листинге 13.7 все выглядит так, словно компонент `About` (не имеющий состояния — то есть функция) не использует `React`. Но при транспиляции этого кода он будет использовать `React` в форме вызовов `React.createElement()`. В главах 1 и 2 объект `React` определялся в форме глобального `window.React`; с модульным (не глобальным) подходом такого не будет. А значит, вам придется определить `React` явно (ch13/router/jsx/about.jsx).

Листинг 13.7. Явное определение React

```
const React = require('react')

module.exports = function About() {
  return <div>
    <a href="http://Node.University" target="_blank">Node.University</a>
      is home to top-notch Node education which brings joy to JavaScript
      ➥ engineers.
  </div>
}
```

В целом проект использует следующую структуру:

```
/router
  /css
    bootstrap.css
    main.css
```

```
  /js
    bundle.js  <-- Собранный (объединенный) файл и его карта исходного кода для удобства отладки
    bundle.js.map
  /jsx
    about.jsx
    app.jsx
    contact.jsx
    content.jsx
    login.jsx
    post.jsx
    posts.jsx
  /node_modules
    index.html
    package.json
    posts.js  <-- Данные для сообщений в блогах (URL, заголовки и текст)
    webpack.config.js
```

Файл index.html содержит минимум информации — он включает только собранный файл.

Листинг 13.8. Файл index.html

```
<!DOCTYPE html>
<html>

  <head>
    <link href="css/bootstrap.css" type="text/css" rel="stylesheet"/>
    <link href="css/main.css" type="text/css" rel="stylesheet"/>
  </head>

  <body>
    <div id="content" class="container"></div>
    <script src="js/bundle.js"></script>
  </body>

</html>
```

Файл webpack.config.js должен содержать как минимум точку входа app.jsx, `babel-loader` и карты исходного кода (ch13/router/webpack.config.js).

Листинг 13.9. Настройка Webpack

```
module.exports = {
  entry: './jsx/app.jsx',
  output: {
    path: __dirname + '/js/',
    filename: 'bundle.js'
  },
  devtool: '#sourcemap',  <-- Задает значение devtool, чтобы видеть правильный мэппинг для исходного кода JSX, а не для транспилированного кода
  stats: {
    colors: true,
    reasons: true
```

```
  },
  module: {
    loaders: [
      {
        test: /\.jsx?$/,
        exclude: /(node_modules)/,
        loader: 'babel-loader'
      }
    ]
  }
}
```

А теперь реализуем макетный компонент `Content`.

13.2.5. Создание макетного компонента

Компонент `Content`, определяемый для родительского объекта `Route`, служит макетом для компонентов About, Posts, Post и Contact. На рис. 13.4 показана схема его реализации.

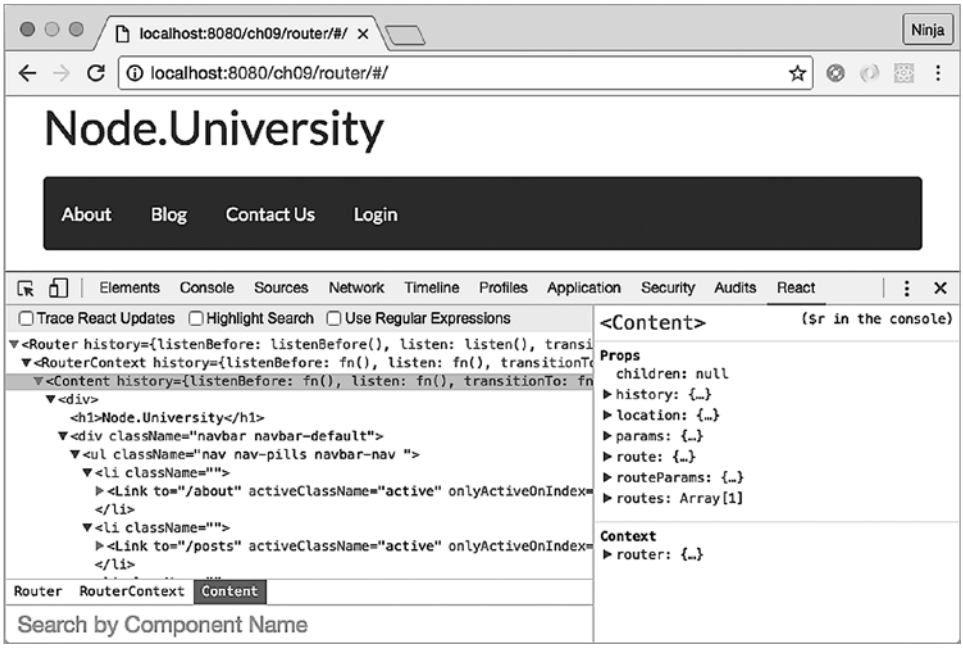


Рис. 13.4. Компонент Content как домашняя страница (без дочерних элементов)

Сначала из React Router импортируются компоненты `React` и `Link` (специальный компонент для рендера навигационных ссылок). `Link` представляет собой обертку

для `<a>`; он содержит некоторые специальные атрибуты, которых нет у нормального якорного тега, например `activeClassName="active"`, который добавляет класс `active`, если этот маршрут активен.

Структура компонента `Content` выглядит примерно так (за исключением нескольких фрагментов, — полный код будет приведен позднее):

```
const React = require('react')
const {Link} = require('react-router')

class Content extends React.Component {
  render() {
    return (
      <div>
      ...
      </div>
    )
  }
}
...
module.exports = Content
```

В `render()` используется замечательная UI-библиотека Twitter Bootstrap (http://getbootstrap.com) для объявления меню с правильными классами. Для создания меню может воспользоваться готовыми классами CSS:

```
<div className="navbar navbar-default">
  <ul className="nav nav-pills navbar-nav ">
    <li ...>
      <Link to="/about" activeClassName="active">
        About
      </Link>
    </li>
    <li ...>
      <Link to="/posts" activeClassName="active">
        Blog
      </Link>
    </li>
    ...
  </ul>
</div>
```

В коде вызывается метод `isActive()`, который возвращает `true` или `false`. Таким образом, активная ссылка в меню будет визуально отличаться от других ссылок:

```
<li className={(this.context.router.isActive('/about'))? 'active': ''}>
  <Link to="/about" activeClassName="active">
    About
  </Link>
</li>
```

Обратите внимание на атрибут `activeClassName` компонента `Link`. Когда этому атрибуту присваивается значение, `Link` применяет класс к активному элементу (выделенная ссылка). Однако стиль необходимо назначить для `<li>`, не только для `Link`. По этой причине мы также вызываем `router.isActive()`.

После определения класса `Content` (полная реализация будет приведена ниже) вы определяете статическое поле/атрибут `contextTypes`, который позволяет использовать `this.context.router`. Если вы используете ES2017+/ES8+[1], статические поля могут поддерживаться, но в ES2015/ES6 или ES2016/ES7 эта возможность отсутствует. Работа над стандартом ES2017/ES8 еще не завершена, но на момент написания книги эта возможность также не поддерживалась. Проверьте текущий список реализованных предложений/возможностей[2] или обратитесь к ES Next (подборка предложений стадии 0).

Этот статический атрибут будет использоваться React Router, чтобы при необходимости библиотека React Router могла заполнить поле `this.context` (из которого можно обратиться к `router.isActive()` и к другим методам):

```
Content.contextTypes = {
  router: React.PropTypes.object.isRequired
}
```

Присваивание `contextType` и `router` значения `required` открывает доступ к `this.context.router.isActive('/about')`, что, в свою очередь, сообщит об активности этого конкретного маршрута.

В листинге 13.10 приведена полная реализация макета `Content`.

Листинг 13.10. Полный код компонента Content

```
const React = require('react')
const {Link} = require('react-router')

class Content extends React.Component {
  render() {
    return (
      <div>
        <h1>Node.University</h1>
        <div className="navbar navbar-default">
          <ul className="nav nav-pills navbar-nav ">
            <li className={(this.context.router.isActive('/about'))?
               ➥ 'active': ''}>
```

[1] За дополнительной информацией о ES2016/ES7 и ES2017/ES8 обращайтесь по адресам https://node.university/blog/498412/es7-es8 и https://node.university/p/es7-es8.

[2] Текущий список предложений на стадии 0–3 и завершенных предложений можно найти в документах TC39 на GitHub: https://github.com/tc39/proposals/blob/master/README.md и https://github.com/tc39/proposals/blob/master/finished-proposals.md.

```
            <Link to="/about" activeClassName="active">
              About
            </Link>
          </li>
          <li className={(this.context.router.isActive('/posts'))?
             ➥ 'active': ''}>
            <Link to="/posts" activeClassName="active">
              Blog
            </Link>
          </li>
          <li className={(this.context.router.isActive('/contact'))?
             ➥ 'active': ''}>
            <Link to="/contact" activeClassName="active">
               Contact Us
            </Link>
          </li>
          <li>
            <Link to="/login" activeClassName="active">
              Login
            </Link>
          </li>
        </ul>
      </div>
      {this.props.children}
    </div>
  )
 }
}
Content.contextTypes = {
  router: React.PropTypes.object.isRequired
}
module.exports = Content
```

Обращается к компоненту Router и его методу для проверки активного маршрута

Использует Link для создания навигационной ссылки

Рендерит дочерние маршруты (определенные в app.jsx)

Определяет, что этому компоненту необходим объект маршрутизатора в контексте

Команда `children` позволяет повторно использовать меню для каждого подмаршрута (маршрут, вложенный в маршрут `/`), такого как `/posts`, `/post`, `/about` и `/contact`:

```
{this.props.children}
```

Рассмотрим другой способ обращения к маршрутизатору из конкретного маршрута, помимо использования `contextTypes`.

13.3. Функциональность React Router

Чтобы больше узнать о возможностях и паттернах использования React Router, рассмотрим другой способ обращения к маршрутизатору из дочерних компонентов и программной навигации по этим компонентам. И конечно, эта глава была бы неполной, если бы мы не рассмотрели разбор параметров URL и передачу данных.

13.3.1. Обращение к маршрутизатору с использованием компонента высшего порядка withRouter

Использование маршрутизатора позволяет среди прочего осуществлять навигацию на программном уровне и обращаться к текущему маршруту. Таким образом, полезно включать в ваши компоненты обращения к маршрутизатору.

Вы уже видели, как обратиться к маршрутизатору из `this.context.router`, задавая статический атрибут класса `contextTypes`:

```
Content.contextTypes = {
  router: React.PropTypes.object.isRequired
}
```

В каком-то смысле вы используете механизм проверки данных для определения API; иначе говоря, компонент должен иметь маршрутизатор. В компоненте `Content` используется именно этот вариант.

Однако `context` зависит от контекста React, а это решение считается экспериментальным; команда React не рекомендует применять его. К счастью, существует и другой способ (по мнению некоторых разработчиков, он проще и лучше; см. http://mng.bz/Xhb9) — `withRouter`.

`withRouter` — компонент высшего порядка (HOC, Higher-Order Component; см. главу 8), который получает компонент в аргументе, внедряет `router` и возвращает другой компонент HOC. Например, `router` внедряется в `Contact` следующим образом:

```
const {withRouter} = require('react-router')
...
<Router ...>
  ...
  <Route path="/contact" component={withRouter(Contact)} />
</Router>
```

Если взглянуть на реализацию компонента `Contact` (функция), объект `router` доступен из свойств (объект-аргумент):

```
const React = require('react')

module.exports = function Contact(props) {
  // props.router - ПОЛЕЗНО!
  return <div>
    ...
  </div>
}
```

Преимущество компонента `withRouter` заключается в том, что он работает с обычными классами React, обладающими состоянием, а также с функциями без состояния.

ПРИМЕЧАНИЕ Хотя React напрямую вроде бы не используется, директива require все равно необходима, потому что этот код будет преобразован в код с командами `React.createElement()`, зависящими от объекта React. За дополнительной информацией обращайтесь к главе 3.

13.3.2. Навигация на программном уровне

Одно из популярных применений `router` — программная навигация: изменение URL (текущей позиции) в коде в зависимости от программной логики, а не действий пользователя. Предположим, у вас имеется приложение, в котором пользователь вводит сообщение в контактной форме, а затем отправляет данные формы. На основании ответа сервера приложение переходит к странице ошибки (`Error`), странице с благодарностью (Thank-You) или странице с информацией (About).

При наличии `router` вы можете выполнить навигацию на программном уровне, в случае необходимости вызвав `router.push(URL)`, где `URL` — определенный путь маршрута. Например, можно перейти к странице About из Contact через 1 секунду.

Листинг 13.11. Вызов router.push() для выполнения перехода

```
const React = require('react')

module.exports = function Contact(props) {
  setTimeout(()=>{props.router.push('about')}, 1000)  ◄——  Переходит через 1 секунду
  return <div>
    <h3>Contact Us</h3>
    <input type="text" placeholder="your email" className="form-control"
     ➥ ></input>
     <textarea type="text" placeholder="your message" className="form-
control">
     ➥ </textarea>
     <button className="btn btn-primary">send</button>
  </div>
}
```

Программная навигация играет важную роль, потому что она позволяет изменить состояние приложения. Теперь посмотрим, как обратиться к параметрам URL, например идентификатору сообщения.

13.3.3. Параметры URL и другие данные маршрутов

Как вы уже видели, `contextTypes` и `router` предоставляют вам объект `this.context.router`. Это экземпляр `<Router/>`, определенный в app.jsx, и он может использоваться для навигации, получения активного пути и т. д. С другой стороны, в `this.props` присутствует и другая интересная информация, причем для обращения к ней статический атрибут не понадобится:

- `history` (считается устаревшим, начиная с v2.x; используйте `context.router`)
- `location`
- `params`
- `route`
- `routeParams`
- `routes`

Объекты `this.props.location` и `this.props.params` содержат данные о текущем маршруте: имя пути, параметры URL (имена, определенные с двоеточием `[:]`) и т. д.

Воспользуемся `params.id` в файле post.jsx, чтобы компонент `Post` в `Array.find()` находил сообщение, соответствующее пути URL, например `router/#/posts/http2` (ch13/router/jsx/post.jsx).

Листинг 13.12. Рендеринг данных сообщения

```
const React = require('react')

module.exports = function Product(props) {
  let post = props.route.posts.find(element=>element.slug ==
  ➥ props.params.id)  ◄——  Находит сообщение по его свойству slug
  return (
    <div>
      <h3>{post.title}</h3>
      <p>{post.text}</p>
      <p><a href={post.link} target="_blank">Continue reading...</a></p>
    </div>
  )
}
```

При переходе к странице Posts (рис. 13.5) выводится список сообщений. Напомню, что определение маршрута выглядит так:

```
<Route path="/posts" component={Posts} posts={posts}/>
```

Щелчок на сообщении вызывает переход к `#/posts/`*ID*. Эта страница повторно использует макет компонента `Content`.

13.3.4. Передача свойств в Router

Часто возникает необходимость в передаче данных вложенным маршрутам, и делается это достаточно просто. В нашем примере компонент `Posts` должен получать данные о сообщениях. В листинге 13.13 компонент `Posts` обращается к свойству, переданному в `<Route/>` в app.jsx: `posts` из файла posts.js. Маршруту можно передать любые данные в атрибуте, например `<Route path="/posts" component={Posts} posts={posts}/>`. После этого вы можете обратиться к данным через `props.route`; например, `props.route.posts` содержит список сообщений.

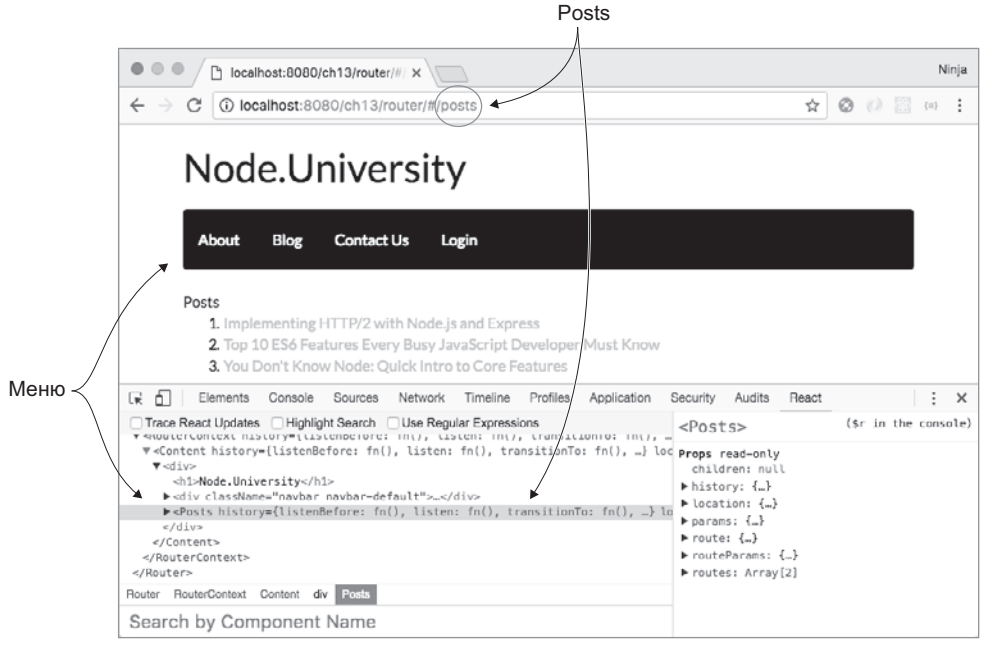


Рис. 13.5. Страница Posts рендерит компонент Posts в компоненте Content (меню), потому что он определяется как дочерний маршрут Content в app.jsx

Листинг 13.13. Реализация Posts с данными из props.route

```
const {Link} = require('react-router')
const React = require('react')

module.exports = function Posts(props) {
  return <div>Posts
    <ol>
    {props.route.posts.map((post, index)=>
      <li key={post.slug}><Link
       ➥ to={`/posts/${post.slug}`} >{post.title}</Link></li>
    )}
    </ol>
  </div>
}
```

Обращается к атрибуту, определенному в объявлении маршрута

Конечно, значением этих данных может быть функция. Таким образом, вы можете передавать обработчики событий компонентам без состояния и реализовать их только в главном компоненте (например, app.jsx).

Все основные части завершены, все готово к запуску проекта! Для этого можно запустить сценарий npm (`npm run build`) или ввести команду `./node_modules/.bin/webpack -w` напрямую. Дождитесь завершения сборки; результат должен выглядеть примерно так:

```
> router@1.0.0 build /Users/azat/Documents/Code/react-quickly/ch13/router
> webpack -w

Hash: 07dc6eca0c3210dec8aa
Version: webpack 1.12.9
Time: 2596ms
        Asset     Size  Chunks             Chunk Names
    bundle.js   976 kB       0  [emitted]  main
bundle.js.map  1.14 MB       0  [emitted]  main
    + 264 hidden modules
```

В новом окне откройте свой любимый статический сервер (я использую node-static, но вы можете создать собственный сервер с использованием Express) и введите адрес перехода в браузере. Попробуйте перейти к `/` и `/#/about`; точный URL будет зависеть от того, запущен ли статический сервер из той же или родительской папки.

> **ПРИМЕЧАНИЕ** Полный исходный код для этого примера здесь не приводится для экономии места. Если вы захотите поэкспериментировать с ним или использовать в качестве заготовки либо предыдущие фрагменты показались вам непонятными вне контекста, полный код доступен по адресу www.manning.com/books/react-quickly или https://github.com/azat-co/react-quickly/tree/master/ch13/router.

13.4. Маршрутизация с использованием Backbone

Если вам нужно организовать маршрутизацию в одностраничном приложении, React легко используется с другими библиотеками маршрутизации или библиотеками в стиле MVC. Например, Backbone — один из самых популярных фреймворков клиентской части со встроенной маршрутизацией URL. Посмотрим, как просто использовать маршрутизатор Backbone для рендера компонентов React. Нужно сделать следующее:

- Определить класс маршрутизатора с объектом `routes` как мэппинг фрагментов URL на функции.
- Выполнить рендеринг элементов React в методах/функциях класса Backbone `Router`.
- Создать экземпляр и запустить объект `Router` из Backbone.

Структура проекта выглядит так:

```
/backbone-router
  /css
    bootstrap.css
    main.css
  /js
    bundle.js
    bundle.map.js
```

```
  /jsx
    about.jsx
    app.jsx
    contact.jsx
    content.jsx
    login.jsx
    post.jsx
    posts.jsx
  /node_modules
    ...
  index.html
  package.json
  posts.js
  webpack.config.js
```

Файл package.json включает Backbone v1.3.3 в дополнение к обычным составляющим, таким как Webpack v2.4.1, React v15.5.4 и Babel v6.11:

```
{
  "name": "backbone-router",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "build": "./node_modules/.bin/webpack -w",
     "i": "rm -rf ./node_modules && npm cache clean && npm install"
  },
  "author": "Azat Mardan",
  "license": "MIT",
  "babel": {
    "presets": [
      "react"
    ]
  },
  "devDependencies": {
    "babel-core": "6.11.4",
    "babel-loader": "6.4.1",
    "babel-preset-react": "6.5.0",
    "backbone": "1.3.3",
    "jquery": "3.1.0",
    "react": "15.5.4",
    "react-dom": "15.5.4",
    "webpack": "2.4.1"
  }
}
```

Исходный код основной логики приложения хранится в файле app.jsx, где выполняются все три из вышеперечисленных задач:

```
const Backbone = require ('backbone')
// Включение других библиотек
```

```
const Router = Backbone.Router.extend({
  routes: {
    '' : 'index',
    'about' : 'about',
    'posts' : 'posts',
    'posts/:id' : 'post',
    'contact' : 'contact',
    'login': 'login'
  },
  ...
})
```

После того как объект `routes` будет определен, можно переходить к определению методов. Значения в `routes` должны использоваться как имена методов:

```
// Включение библиотек
const Router = Backbone.Router.extend({
  routes: {
    '' : 'index',
    'about' : 'about',
    'posts' : 'posts',
    'posts/:id' : 'post',
    'contact' : 'contact',
    'login': 'login'
  },
  index: function() {
    ...
  },
  about: function() {
    ...
  }
  ...
})
```

Каждый фрагмент URL распределяется на функцию. Например, `#/about` соответствует `about`. Таким образом, вы можете определить эти функции и рендерить ваши компоненты React из них. Данные будут передаваться в свойстве (`router` или `posts`):

```
const {render} = require ('react-dom')
// ...
const Router = Backbone.Router.extend({
  routes: {
    ...
  },
  index: function() {
    render(<Content router={router}/>, content)
  },
  about: function() {
    render(<Content router={router}>
      <About/>
    </Content>, content)
```

Использует деструктуризацию для импортирования и определения render() из ReactDOM.render()

Создает Content с внутренним компонентом About. Маршрутизатор может передаваться в свойстве

```
  },
  posts: function() {
    render(<Content>
      <Posts posts={posts}/>
    </Content>, content)
  },
  post: function(id) {
    render(<Content>
      <Post id={id} posts={posts}/>
    </Content>, content)
  },
  contact: function() {
    render(<Content>
      <Contact />
    </Content>, content)
  },
  login: function() {
    render(<Login />, content)
  }
})

let router = new Router()

Backbone.history.start()
```

Передает Post необходимые данные, такие как параметр URL (id), и отправляет данные

Выполняет прорисовку Login без Content

Создает экземпляр Router и запускает историю браузера

Переменная `content` содержит узел DOM (она должна объявляться до маршрутизатора):

```
let content = document.getElementById('content')
```

По сравнению с примером для React Router вложенные компоненты (такие, как `Post`) получают свои данные не в `props.params` или `props.route.posts`, а в `props.id` и `props.posts`. На мой взгляд, от этого все становится более понятным — а это всегда хорошо. С другой стороны, воспользоваться декларативным синтаксисом JSX не удастся, и приходится использовать более императивный стиль.

Полный код этого проекта доступен по адресам www.manning.com/books/react-quickly и https://github.com/azat-co/react-quickly/tree/master/ch13/backbone-router. Этот пример поможет вам взяться за дело, если вы уже работаете в Backbone или собираетесь перейти на Backbone. И даже если вы не планируете использовать Backbone, он показывает, как хорошо React сочетается с другими библиотеками.

13.5. Вопросы

1. Для React Router версии 2.*x* (рассмотренной в этой главе) необходимо предоставить собственную реализацию истории, потому что реализации по умолчанию не существует. Да или нет?

2. Какая реализация истории лучше поддерживается старыми браузерами: история идентификатора фрагмента или браузерная история `pushState` в HTML5?
3. Что необходимо реализовать, чтобы иметь доступ к объекту `router` в компоненте маршрута при использовании React Router версии 2.*x*?
4. Как обратиться к параметрам URL в компоненте маршрута (с состоянием или без) при использовании React Router версии 2.*x*?
5. React Router требует использования Babel и Webpack. Да или нет?

13.6. Итоги

- Вы можете самостоятельно реализовать «наивную» маршрутизацию в React, прослушивая `hashchange`.
- React Router предоставляет синтаксис JSX для определения иерархии маршрутизации: `<Router><Route/></Router>`.
- Вложенные маршруты не обязаны иметь URL, вложенные по отношению к своим родительским маршрутам; пути не зависят от вложенности.
- Чтобы использовать историю идентификатора фрагмента, присвойте `queryKey` значение `false`.
- При использовании JSX необходимо включать React (`require('react')`) даже в том случае, если React напрямую не используется. Дело в том, что JSX преобразуется в вызовы `React.createElement()`, для которых необходима поддержка React.

13.7. Ответы

1. Да. Версия 1.*x* React Router загружала реализацию истории по умолчанию; но в версии 2.*x* необходимо передать соответствующую библиотеку (либо из автономного пакета, либо привлекаемую к библиотеке маршрутизации).
2. История идентификатора фрагмента лучше поддерживается старыми браузерами.
3. Статический атрибут класса `contextTypes` с обязательным объектом `router`.
4. Из `props.params` или `props.routeParams`.
5. Нет. Вы можете использовать React Router в «чистом» виде и/или с другими средствами сборки, такими как Gulp и Browserify.

14 Работа с данными с использованием Redux

Посмотрите вступительный видеоролик к этой главе, отсканировав QR-код или перейдя по ссылке http://reactquickly.co/videos/ch14.

ЭТА ГЛАВА ОХВАТЫВАЕТ СЛЕДУЮЩИЕ ТЕМЫ:

- Поддержка однонаправленной передачи данных в React.
- Архитектура данных Flux.
- Использование библиотеки данных Redux.

До настоящего момента мы использовали React для создания пользовательских интерфейсов — это самый распространенный сценарий использования React. Однако многие пользовательские интерфейсы должны работать с данными. Эти данные поступают от сервера (серверная часть) или от другого компонента в браузере.

В области работы с данными React предоставляет много возможностей:

- *Интеграция с фреймворками в стиле MVC* — этот вариант идеально подходит, если вы уже используете или собираетесь использовать фреймворк в стиле MVC для одностраничных приложений (например, Backbone и модели Backbone).
- *Написание собственного метода данных или библиотеки* — этот вариант хорошо подходит для небольших UI-компонентов (например, для получения списка банковских счетов в таблице).
- *Использование стека React* — этот вариант обеспечивает максимальную совместимость (интеграция кода с минимальными проблемами) и в наибольшей степени соответствует философии React.

В этой главе рассматривается одна из самых популярных разновидностей третьего подхода — Redux. Начнем с рассмотрения передачи данных в компонентах React.

ПРИМЕЧАНИЕ Также существует архитектура Flux и библиотека flux от Facebook. Я продемонстрирую Redux вместо библиотеки flux, потому что Redux более активно используется в проектах. flux скорее служит для подтверждения работоспособности концепции архитектуры Flux, которую реализует Redux. Вы можете рассматривать Redux и flux (библиотеку) как две реализации архитектуры Flux. (Я буду рассматривать архитектуру Flux, но не библиотеку.)

ПРИМЕЧАНИЕ Исходный код примеров этой главы доступен по адресам www.manning.com/books/react-quickly и https://github.com/azat-co/react-quickly/tree/master/ch14 (в папке ch14 репозитория GitHub https://github.com/azat-co/react-quickly). Также некоторые демонстрационные примеры доступны по адресу http://reactquickly.co/demos.

14.1. Поддержка однонаправленной передачи данных в React

React — уровень представления, предназначенный для работы с однонаправленной передачей данных (рис. 14.1). Паттерн *однонаправленной передачи данных* (или *одностороннее связывание*) существует там, где между составляющими нет изменяемых (двусторонних) ссылок (под «составляющими» в данном случае понимаются части с разной функциональностью). Например, представление и модель не могут иметь двусторонних ссылок. Вскоре мы вернемся к двусторонней передаче данных.

Например, если у вас имеется модель банковского счета и представление банковского счета, данные могут передаваться только от модели к представлению, но не наоборот. Другими словами, изменения в модели могут вызывать изменения в представлении (рис. 14.2). Здесь очень важно понимать, что представления не могут изменять модели напрямую.

Однонаправленная передача данных гарантирует, что для каждого ввода для ваших компонентов вы получите один и тот же прогнозируемый результат: выражение `render()`. Паттерн React резко контрастирует с двусторонним, двунаправленным паттерном связывания в Angular.js и других фреймворках.

Например, в двунаправленной передаче данных изменения в модели могут вызывать изменения в представлениях, а изменения в представлениях (пользовательский ввод) могут вызывать изменения в моделях. По этой причине с двунаправленной передачей данных состояние представления становится менее предсказуемым, что усложняет понимание кода, его отладку и сопровождение (рис. 14.3). Главное — помнить, что представления могут напрямую изменять модели, что принципиально отличается от однонаправленной передачи данных.

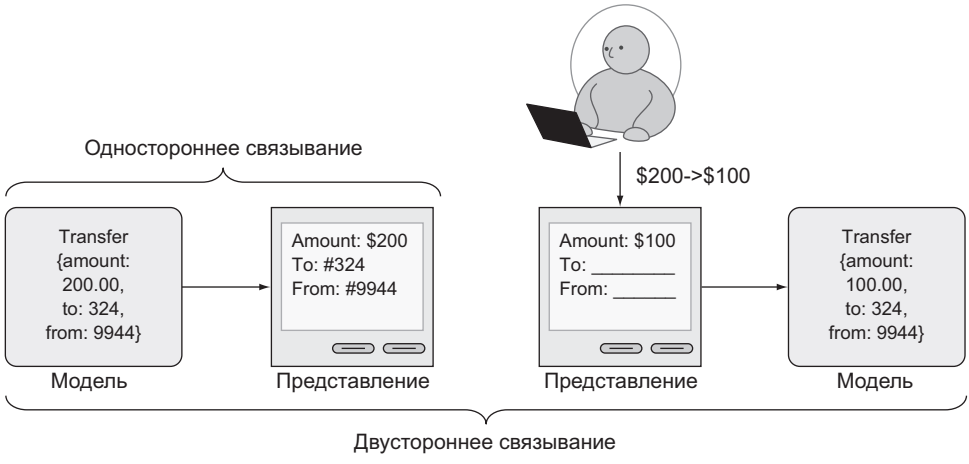


Рис. 14.1. Односторонняя и двусторонняя передача данных

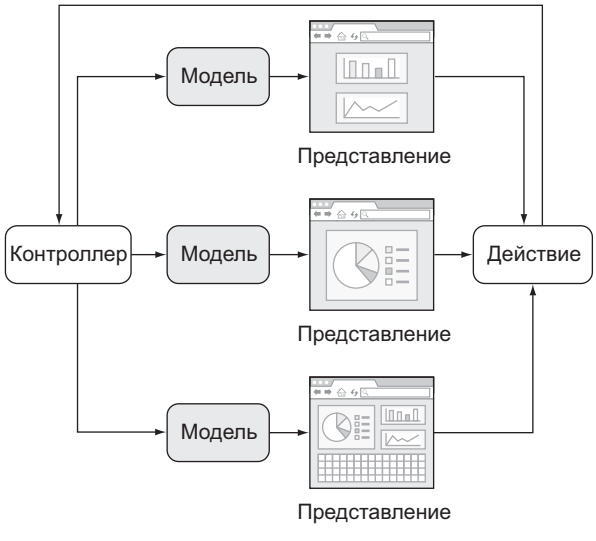


Рис. 14.2. Упрощенное представление однонаправленной передачи данных, при которой представления не могут изменять модель напрямую

Интересно, что двунаправленную передачу данных (*двустороннее связывание*) некоторые разработчики Angular считают преимуществом. Не углубляясь в споры, скажу, что с двунаправленной передачей приходится писать меньше кода.

Допустим, имеется поле ввода вроде изображенного на рис. 14.1. Все, что от вас требуется, — определить переменную в шаблоне; когда пользователь вводит данные в поле, значение будет обновляться в модели. В то же время значение на веб-

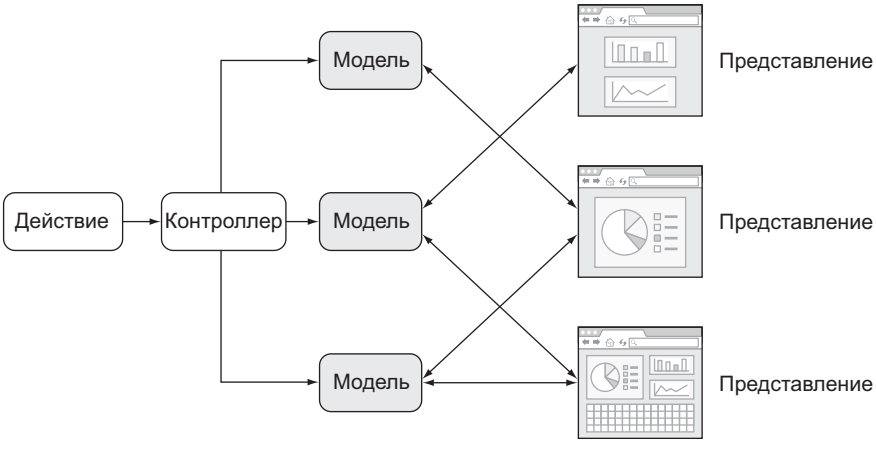


Рис. 14.3. Упрощенная схема двунаправленной передачи данных, типичной для архитектуры в стиле MVC

странице будет обновляться при изменении модели (в результате запроса XHR GET, например). Таким образом, изменения возможны в двух направлениях: от представления к модели и от модели к представлению. Такой подход прекрасно подходит для построения прототипа, но не блещет в сложных пользовательских интерфейсах в том, что касается быстродействия, отладки, масштабирования в ходе разработки и т. д. На первый взгляд это утверждение кажется спорным — но не спешите.

Я построил много сложных UI-приложений на базе фреймворков MVC и MVW с двунаправленной передачей данных, и они справляются со своим делом. В двух словах: проблемы возникают из-за того, что разные представления могут манипулировать с разными моделями, и наоборот. Если вы используете одну-две изолированные модели и представления, это нормально; но чем больше приложение, тем больше моделей и представлений обновляют друг друга. Становится все труднее понять, почему некоторая модель или представление находятся в конкретном состоянии, потому что вы не можете легко определить, какие модели/представления и в каком порядке обновили их. Возникают огромные проблемы с отслеживанием, а также с поиском ошибок. Вот почему многие разработчики недовольны двунаправленной передачей данных во фреймворках MVC (например, Angular); они считают, что это антипаттерн, который усложняет отладку и масштабирование.

С другой стороны, при однонаправленной передаче данных модель обновляет представление — и все. Дополнительно однонаправленная передача данных позволяет выполнять рендеринг на стороне сервера, потому что представления становятся неизменяемой функцией состояния (изоморфный/универсальный JavaScript).

А пока просто учтите, что однонаправленная передача данных — один из главных аргументов в пользу React:

- Удобочитаемость кода и логичность происходящего благодаря одному источнику изменений (состояние/модель → представление).

- Удобство отладки кода с возможностью возврата к предыдущему состоянию кода[1]: например, можно легко отправить на сервер дамп с информацией об исключениях и ошибках.
- Рендеринг на стороне сервера без консольного браузера: изоморфный[2], или универсальный[3], JavaScript.

Могу поделиться своим личным опытом работы с Angular, если вам интересно. Я лишь немного работал с Angular 1, потому что считал эту версию неудовлетворительной, но потом я прошел курсы по Angular 2 и понял, как я ошибался. Я исправил свою ошибку и теперь полностью избегаю любого кода Angular.

14.2. Архитектура данных Flux

Flux (https://facebook.github.io/flux) — архитектурный паттерн для передачи данных, разработанный Facebook для использования в приложениях React. Главные особенности Flux — однонаправленная передача данных и избавление от сложности паттернов в стиле MVC.

А теперь рассмотрим типичный паттерн в стиле MVC, изображенный на рис. 14.4. Действия инициируют события в контроллере, который работает с моделями. Затем, согласно моделям, приложение выполняет рендер представлений, и тут начинается страшное. Каждое представление обновляет модели — не только свою собственную модель, но и другие модели, а модели обновляют представления (двунаправленная передача данных). В этой архитектуре легко запутаться, она создает слишком много проблем с пониманием и отладкой.

И наоборот, Flux предлагает использовать однонаправленную передачу данных (рис. 14.5). В этом случае действия из представлений проходят через диспетчер, который, в свою очередь, обращается к хранилищу данных. (Flux — не просто новая терминология, а замена для MVC.) Хранилище отвечает за данные и отображение информации в представлениях. Представления не изменяют данные, но содержат действия, которые снова проходят через диспетчер.

Однонаправленная передача данных упрощает тестирование и отладку. Более подробная схема архитектуры Flux изображена на рис. 14.6.

Сначала появилась архитектура Flux. Лишь позднее команда Facebook выпустила модуль `flux` (www.npmjs.com/package/flux, https://github.com/facebook/flux), который может использоваться с React для реализации Flux. Модуль `flux` скорее служит для подтверждения работоспособности концепции архитектуры Flux, и разработчики React редко пользуются им.

---

[1] Dan Abramov, «Live React: Hot Reloading with Time Travel» (презентация, ReactEurope 2015), http://mng.bz/uSxq.

[2] Spike Brehm, «Isomorphic JavaScript: The Future of WebAir Apps», Airbnb Engineering & Data Science, 11 ноября 2013 г., http://mng.bz/i34M.

[3] Michael Jackson, «Universal JavaScript», 8 июня 2015 г., http://mng.bz/7GXE.

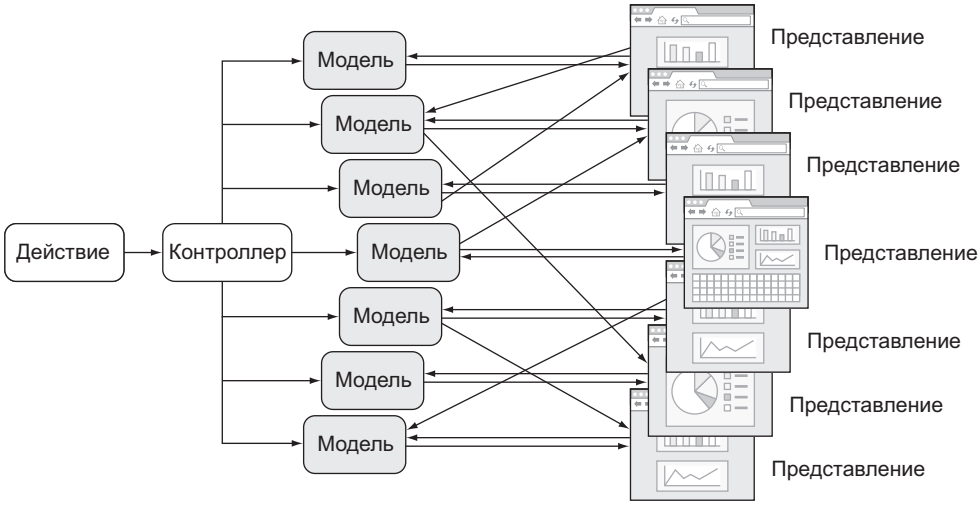


Рис. 14.4. Архитектура в стиле MVC создает лишнюю сложность: любое представление может инициировать изменения в любой модели, и наоборот

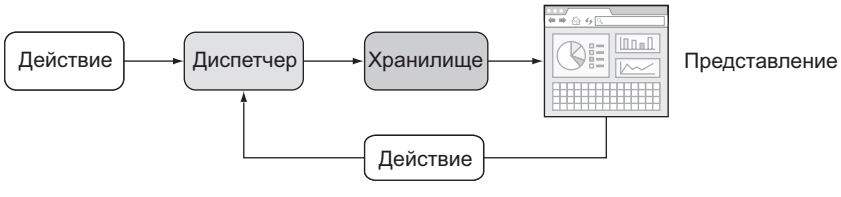


Рис. 14.5. Архитектура Flux упрощает передачу данных, которая может проходить только в одном направлении (от хранилища к представлению)

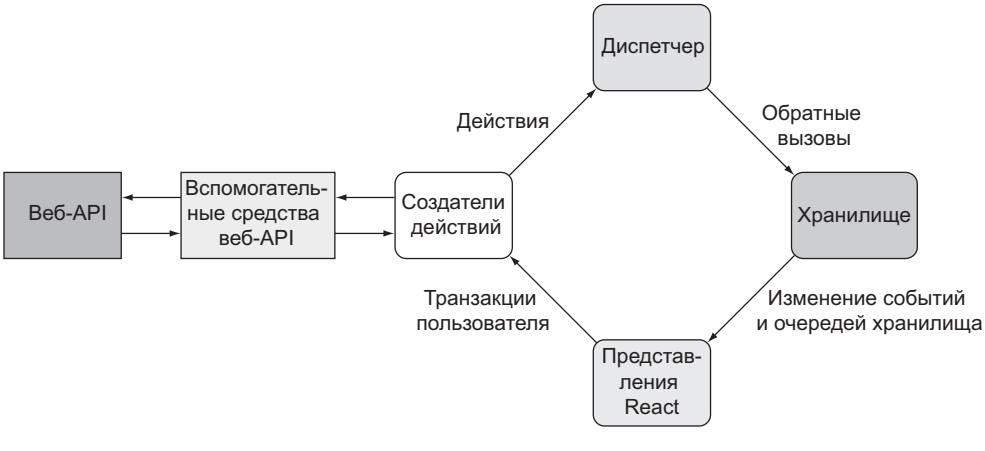


Рис. 14.6. Архитектура Flux: действия активизируют диспетчер, инициирующий операции хранилища, которые выполняют рендеринг представлений

СОВЕТ Нет смысла повторять слова великих умов, уже высказывавшихся по поводу Flux. Я рекомендую посмотреть видеоролик «Hacker Way: Rethinking Web App Development at Facebook» с официального сайта Flux: http://mng.bz/wygf.

Лично меня архитектура Flux приводит в замешательство — и не меня одного. Существует много реализаций Flux, включая Redux, Reflux и другие библиотеки. Первые читатели этой книги по программе раннего доступа Manning знают, что в первую версию было включено описание Reflux, но я убрал его из этой версии. Из моего неофициального источника, статьи Дэвида Уоллера (David Waller) «React.js architecture — Flux vs. Reflux» (http://mng.bz/5GHx), и подтвержденных данных загрузок npm следует, что Redux популярнее Flux и Reflux. В этой книге я использую библиотеку Redux; некоторые разработчики отдают ей предпочтение перед Flux.

14.3. Использование библиотеки данных Redux

Redux (`redux`, www.npmjs.com/package/redux) — одна из самых популярных реализаций архитектуры Flux. Основными достоинствами Redux являются:

- *Богатая экосистема* — см., например, Awesome Redux (https://github.com/xgrommx/awesome-redux).
- *Простота* — не требуется ни диспетчер, ни регистрация в хранилище, а минимальная версия занимает всего 99 строк кода (http://mng.bz/00Ap).
- *Удобство для разработчика* — например, вы можете выполнять оперативную перезагрузку с возвратом во времени (см. Live React: Hot Reloading with Time Travel», http://mng.bz/uSxq).
- *Минимализм* — например, компонент высшего порядка для отмены/повтора требует минимального объема кода (https://github.com/omnidan/redux-undo).
- Поддержка рендеринга на стороне сервера.

Я не буду тратить время на подробные объяснения того, почему Redux лучше Flux. Если вас заинтересует эта тема, ознакомьтесь с мнением автора Redux: «Why Use Redux over Facebook Flux?» по адресу http://mng.bz/z9ok.

Redux — автономная библиотека, реализующая контейнер состояния, — что-то вроде гигантской переменной, содержащей все данные, с которыми работает ваше приложение, хранилища и изменения во время выполнения. Redux можно использовать автономно или на сервере. Как упоминалось ранее, Redux также часто используется в сочетании с React; эта комбинация реализована в другой библиотеке, `react-redux` (https://github.com/reactjs/react-redux).

При использовании Redux в приложениях React участвуют некоторые составляющие:

- Хранилище, которое содержит все данные и предоставляет методы для работы с этими данными. Хранилище создается функцией `createStore()`.

- Компонент `Provider`, который позволяет любому компоненту получить данные из хранилища.
- Функция `connect()`, которая инкапсулирует любой компонент и позволяет отобразить отдельные части состояния приложения из хранилища на свойства компонента.

Снова взгляните на схему архитектуры Flux на рис. 14.5: понятно, почему на ней присутствует хранилище. Единственный способ изменения внутреннего состояния — передача действия, а действия относятся к хранилищу.

Все изменения в хранилище выполняются посредством *действий* (actions). Каждое действие сообщает вашему приложению, что произошло и какую часть хранилища следует изменить. Действия также предоставляют данные; разумеется, это полезно, потому что каждое приложение работает с изменяющимися данными.

Способ изменения данных в хранилище задается *редьюсерами* (reducers), которые представляют собой «чистые» функции с сигнатурой (*состояние, действие*) → *состояние*. Другими словами, применяя действие к текущему состоянию, вы получаете новое состояние. Тем самым обеспечивается предсказуемость и возможность возврата (функцией отмены и средствами отладки) к предыдущим состояниям.

Вот как выглядит файл редьюсера для приложения списка текущих дел, в котором `SET_VISIBILITY_FILTER` и `ADD_TODO` являются действиями:[1]

```
function todoApp(state = initialState, action) {
  switch (action.type) {
    case 'SET_VISIBILITY_FILTER':
      return Object.assign({}, state, {
        visibilityFilter: action.filter
      })
    case 'ADD_TODO':
      return Object.assign({}, state, {
        todos: [
          ...state.todos,
          {
            text: action.text,
            completed: false
          }
        ]
      })
    default:
      return state
  }
}
```

Определяет действие

Применяет редьюсер для создания нового состояния посредством копирования[1] текущего состояния и значений visibilityFilter

Определяет действие ADD_TODO

Применяет редьюсер для создания нового состояния посредством копирования текущего состояния и новых значений TODO «text» и «completed» в последний элемент массива todos

Определяет вариант по умолчанию, который в данном случае возвращает текущее состояние

Ваше приложение Redux может содержать один или несколько редьюсеров (хотя может не содержать ни одного). Каждый раз, когда вы вызываете действие, вызывается каждый редьюсер. Редьюсеры отвечают за изменение данных в хранилище;

[1] Object.assign(), http://mng.bz/O6pl.

вот почему вы должны быть внимательны с тем, что они делают во время некоторых типов действий.

Как правило, редьюсер представляет собой функцию с состоянием и действиями-аргументами. Например, возможным действием может быть «загрузка видео» (`FETCH_MOVIE`), которое может быть выполнено при помощи редьюсера. Код действия описывает, каким образом действие преобразует состояние в следующее состояние (добавляет видео в состояние). Функция-редьюсер содержит огромную конструкцию `switch/case` для обработки действий. Однако существует удобная библиотека, которая делает редьюсеры более функциональными и — сюрприз! — удобочитаемыми. Эта библиотека называется `redux-actions`, и вы увидите, как использовать ее вместо `switch/case`.

> **СОВЕТ** Создатель Redux Дэн Абрамов (Dan Abramov (https://github.com/gaearon) рекомендует для расширения кругозора почитать о Redux следующие статьи: «Why Use Redux Over Facebook Flux?» (http://mng.bz/9syg) и «What Could Be the Downsides of Using Redux Instead of Flux» (http://mng.bz/Ux9l).

14.3.1. Клон Netflix на базе Redux

Всем нам нравятся старые добрые голливудские фильмы, верно? Давайте напишем приложение, которое выводит список классических фильмов, то есть клон Netflix (но только домашнюю страницу — без потоковой передачи или чего-нибудь в этом роде). Приложение выводит таблицу фильмов (рис. 14.7), а при щелчке на изображении открывается подробное описание (рис. 14.8).

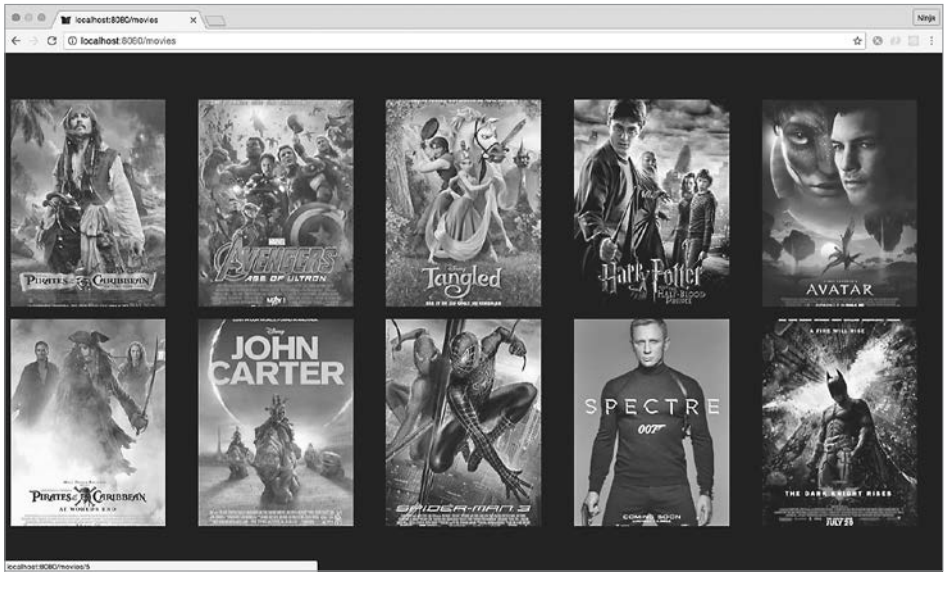


Рис. 14.7. Клон Netflix с галереей фильмов на домашней странице

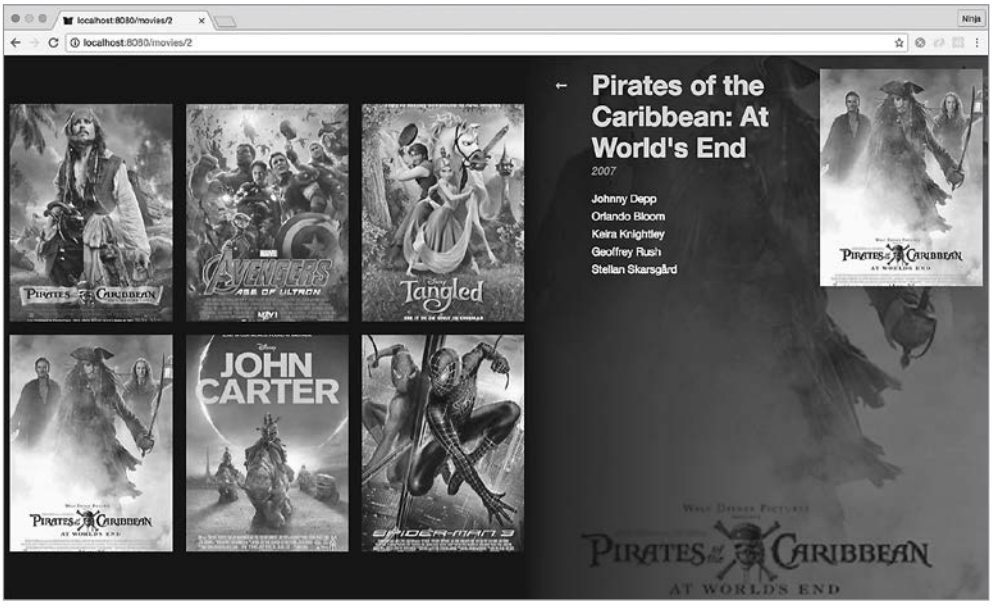


Рис. 14.8. При щелчке на постере выводится подробная информация о фильме

Основная задача этого примера — научить вас применять Redux в реальных ситуациях для передачи данных компонентам React. Для упрощения эти данные будут читаться из файла JSON. При выводе подробной информации о фильмах будет использоваться библиотека React Router, о которой вы узнали в предыдущей главе.

Проект содержит три компонента: `App`, `Movies` и `Movie`. У каждого компонента имеется файл CSS, и для улучшения структуры кода каждый компонент находится в отдельной папке (рекомендуемый способ инкапсуляции компонентов React вместе со стилями). Структура проекта выглядит так:

```
/redux-netfix
  /build  <-- Каталог build, в который Webpack будет записывать пакеты
    index.js
    styles.css
  /node_modules
    ...
  /src
    /components
      /app  <-- Каталог app для макетного компонента
        app.css
        app.js
      /movie  <-- Каталог movies для таблицы с фильмами
        movie.css
        movie.js
      /movies  <-- Каталог movie для компонентов представления отдельного фильма
```

```
    movies.css
    movies.js
  /modules
    index.js   <-- Файл, который объединяет редьюсеры и предоставляет доступ к ним
    movies.js  <-- Редьюсеры Redux для получения данных фильмов и отдельного фильма
  index.js     <-- Точка входа проекта, определяющая провайдер Redux и редьюсеры
  movies.json  <-- Данные фильмов
  routes.js    <-- Маршруты React Router
index.html
package.json
webpack.config.json
```

Разобравшись со структурой каталогов проекта, обратимся к зависимостям и конфигурациям сборки.

14.3.2. Зависимости и конфигурации

Для проекта необходимо создать ряд зависимостей. Webpack (https://github.com/webpack/webpack) собирает все файлы для непосредственного использования, а дополнительный плагин `extract-text-webpack-plugin` собирает стили из нескольких включений `<style>` в один файл style.css. Также в проекте используются загрузчики Webpack:

- `json-loader`
- `style-loader`
- `css-loader`
- `babel-loader`

К числу других модулей зависимостей разработки проекта относятся:

- Babel (https://github.com/babel/babel) и его конфигурации транспилируют ECMAScript 6 в широко поддерживаемый браузерами традиционный код JavaScript (он же ECMAScript 5): `babel-polyfill` эмулирует полную среду ES2015, `babel-preset-es2015` предназначается для ES6/ES2015, `babel-preset-stage-0` предоставляет новые современные возможности ES7+, а `babel-preset-react` используется для JSX.
- `react-router` (https://github.com/reactjs/react-router) отображает иерархию компонентов на основании их текущего местоположения. Этот модуль также помогает упорядочить компоненты в иерархию на основании URL.
- `redux-actions` (https://github.com/acdlite/redux-actions) создает структуру преобразователей.
- ESLint (http://eslint.org) и его основной плагин обеспечивают правильный стиль JavaScript/JSX.

- `concurrently` (www.npmjs.com/package/concurrently) — инструмент Node для организации параллельного выполнения процессов (например, сборок Webpack).

Файл package.json содержит список всех зависимостей, конфигураций Babel и сценариев `npm`; он должен содержать как минимум данные, приведенные в листинге 14.1 (ch14/redux-netflix/package.json). Как обычно, модули можно установить вручную командой `npm i NAME`, ввести `package.json` и выполнить команду `npm i` или скопировать файл package.json и выполнить команду `npm i`. Проследите за тем, чтобы использовались именно те версии библиотек, которые указаны в package.json; в противном случае работа кода может быть нарушена.

Листинг 14.1. Зависимости клона Netflix

```
{
  "name": "redux-netflix",
  "version": "0.0.1",
  "description": "A sample project in React and Redux that copies Netflix's
  ➥ features and workflow",
  "main": "./build/index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "start": "concurrently \"webpack --watch --config webpack.config.js\"
     ➥ \"webpack-dev-server\""
  },
  "repository": {
    "type": "git",
    "url": "git+https://github.com/azat-co/react-quickly.git"
  },
  "author": "Azat Mardan (http://azat.co)",
  "license": "MIT",
  "bugs": {
    "url": "https://github.com/azat-co/react-quickly/issues"
  },
  "homepage": "https://github.com/azat-co/react-quickly#readme",
  "devDependencies": {
    "babel-core": "6.11.4",
    "babel-eslint": "6.1.2",
    "babel-loader": "6.2.4",
    "babel-polyfill": "6.9.1",
    "babel-preset-es2015": "6.9.0",
    "babel-preset-react": "6.11.1",
    "babel-preset-stage-0": "6.5.0",
    "concurrently": "2.2.0",
    "css-loader": "0.23.1",
    "eslint": "3.1.1",
    "eslint-plugin-babel": "3.3.0",
    "eslint-plugin-react": "5.2.2",
    "extract-text-webpack-plugin": "1.0.1",
    "json-loader": "0.5.4",
    "style-loader": "0.13.1",
    "webpack": "1.13.1",
```

Определяет сценарий для построения и запуска Webpack Dev Server с использованием инструмента concurrency

Устанавливает различные модули, загрузчики и плагины Babel

Устанавливает concurrently для ускорения запуска сценариев npm

Устанавливает extract-text-webpack-plugin для объединения встроенных стилей в один пакет

```
    "webpack-dev-server": "1.14.1"
    "react": "15.2.1",
    "react-dom": "15.2.1",
    "react-redux": "4.4.5",   <-- Устанавливает react-redux для работы с данными
    "react-router": "2.6.0",
    "redux": "3.5.2",
    "redux-actions": "0.10.1"   <-- Устанавливает redux-actions для организации
  }                                 преобразователей Redux
}
```

Так как Webpack используется для сборки зависимостей, все необходимые пакеты находятся в bundle.js. По этой причине все зависимости размещаются в `devDependencies`. (Я весьма разборчив в выборе файлов для развертывания — не хочу, чтобы неиспользуемые модули просто занимали место и создавали угрозы для безопасности.) npm игнорирует `devDependencies` при использовании флага `--production`, как в `npm i --production`.

Теперь определим процесс построения в файле webpack.config.js (ch14/redux-netflix/webpack.config.js).

Листинг 14.2. Конфигурационный файл Webpack для клона Netflix

```
const path = require('path')
const ExtractTextPlugin = require('extract-text-webpack-plugin')

module.exports = {
  entry: {
    index: [
      'babel-polyfill',   <-- Применяет polyfill для полной эмуляции среды ES2015
      './src/index.js'    <-- Задает точку входа (не обязана быть файлом *.jsx)
    ]
  },
  output: {
    path: path.join(__dirname, 'build'),   <-- Задает выходной каталог с path.join() для большей надежности при кросс-платформенном использовании (например, в системе Windows)
    filename: '[name].js'
  },
  target: 'web',
  module: {
    loaders: [{   <-- Применяет загрузчики, определенные в массиве
      loader: 'babel-loader',
      include: [path.resolve(__dirname, 'src')],
      exclude: /node_modules/,
      test: /\.js$/,
      query: {
        presets: ['react', 'es2015', 'stage-0']   <-- Задает конфигурации Babel (то есть что нужно сделать с кодом)
      }
    }, {
      loader: 'json-loader',   <-- Применяет загрузчик JSON для имитации базы данных в файлах JSON
      test: /\.json$/
    }, {
```

```
      loader: ExtractTextPlugin.extract('style',
        'css?modules&localIdentName=[local]__[hash:base64:5]'),
      test: /\.css$/,
      exclude: /node_modules/
    }]
  },
  resolve: {
    modulesDirectories: [
      './node_modules',
      './src'
    ]
  },
  plugins: [
    new ExtractTextPlugin('styles.css')
  ]
}
```

Применяет загрузчик из плагина для извлечения стилей и объединения их в один файл (вместо многих файлов)

Предоставляет плагин для извлечения текста

Довольно разговоров о конфигурации. В следующем разделе мы начнем работать с Redux.

14.3.3. Включение Redux

Чтобы технология Redux работала в приложении React, на верхнем уровне иерархии компонентов должен находиться компонент `Provider`. Этот компонент находится в пакете `react-redux` и внедряет данные из хранилища в компоненты. Да, все верно: использование `Provider` в качестве компонента верхнего уровня означает, что все дочерние компоненты будут иметь доступ к хранилищу. Удобно!

Для работы компонента `Provider` необходимо передать хранилище в его свойстве `store`. Объект `Store` представляет состояние приложения. Redux (`react-redux`) предоставляет функцию `createStore()`, которая получает редьюсер(-ы) из ch14/redux-netflix/scr/modules/index.js и возвращает объект `Store`.

Для рендера компонента `Provider` и всего его поддерева компонентов используется метод `react-dom render()`. Он получает первый аргумент (`<Provider>`) и подключает его к элементу, передаваемому во втором аргументе (`document.getElementById('app')`).

В итоге точка входа вашего приложения должна выглядеть примерно так, как показано в листинге 14.3 (ch14/redux-netflix/index.js). При определении компонента `Provider` передается экземпляр `Store` (с редьюсерами) в формате JSX.

Листинг 14.3. Главная точка входа приложения

```
const React = require('react')
const { render } = require('react-dom')
const { Provider } = require('react-redux')
const { createStore } = require('react-redux')
const reducers = require('./modules')
```

```
const routes = require('./routes')

module.exports = render((
  <Provider store={createStore(reducers)}>
    {routes}
  </Provider>
), document.getElementById('app'))
```

Чтобы все приложение могло использовать возможности Redux, необходимо реализовать код в дочерних компонентах, например код установления связи с хранилищем. Функция `connect()` из того же пакета `react-redux` получает несколько аргументов и возвращает функцию, которая инкапсулирует ваш компонент, чтобы некоторые части хранилища были отражены в его свойствах. Вскоре вы увидите, как это происходит.

С index.js работа завершена. Компонент `Provider` обеспечивает передачу данных из хранилища во все подключенные компоненты, так что передавать свойства напрямую не нужно. Однако некоторые части, в частности маршруты, редьюсеры и действия, все еще отсутствуют. Давайте последовательно рассмотрим их.

14.3.4. Маршруты

`react-router` позволяет объявить иерархию компонентов для каждого пути в браузере. Библиотека React Router рассматривалась в главе 13, так что она должна быть вам знакома; там она применялась для маршрутизации на стороне клиента. Маршрутизация React не имеет четкой связи с маршрутами на стороне сервера, но иногда ее можно применять для этой цели, особенно в сочетании с приемами, описанными в главе 16.

Вся суть React Router заключается в том, что каждый маршрут может быть объявлен парой вложенных компонентов `Route`, каждый из которых получает два свойства:

- `path` — URL-путь, который может содержать параметры URL, например `/movies:/id for localhost:8080/movies/1021`. Использование `/` позволяет определить путь независимо от пути родительского маршрута, например `/:id` для `localhost:8080/1012`.
- `component` — ссылка на компонент, который будет рендериться при переходе пользователя по соответствующему пути. Также будут отрендерены все родительские компоненты до `Provider`. Например, при переходе к `localhost:8080/movies/1021` в листинге 14.4 будут отрендерены `Movie`, `Movies` и `App`.

Подборка постеров должна отображаться как в корне, так и в `/movies`. Кроме того, по маршруту `/movies/:id` должна выводиться подробная информация о выбранном фильме. Конфигурация маршрута использует компонент `IndexRoute` из листинга 14.4 (ch14/redux-netflix/src/routes.js).

Листинг 14.4. Определение маршрутизации URL с React Router

```
const React = require('react')
const {
  Router,
  Route,
  IndexRoute,
  browserHistory
} = require('react-router')
const App = require('components/app/app.js')
const Movies = require('components/movies/movies.js')
const Movie = require('components/movie/movie.js')

module.exports = (
  <Router history={browserHistory}>
    <Route path="/" component={App}>
      <IndexRoute component={Movies} />
      <Route path="movies" component={Movies}>
        <Route path=":id" component={Movie} />
      </Route>
    </Route>
  </Router>
)
```

Предоставляет маршрутизатору историю браузера или идентификатора фрагмента

Определяет параметр URL с идентификатором фильма (:id)

Определяет индексный маршрут, то есть маршрут для пустого URL /

Как `IndexRoute`, так и `Route` вложен в маршрут верхнего уровня. Таким образом, компонент Movies рендерится как для корня, так и для пути `/movies`. Представлению конкретного фильма необходим идентификатор для получения данных этого фильма из хранилища Redux, поэтому мы определяем путь с параметром URL. Для этого используется синтаксис с двоеточием — `path=":id"`. На рис. 14.9 показано, как это представление и его URL выглядят на малом экране (благодаря адаптируемой разметке CSS). Обратите внимание на URL `movies/8`: здесь 8 — идентификатор фильма. Теперь посмотрим, как организовать выборку данных с использованием редьюсеров Redux.

14.3.5. Объединение редьюсеров

Рассмотрим модули, к которым применяется функция `createStore()` из src/index.js:

```
...
const reducers = require('./modules')
...

module.exports = render((
  <Provider store={createStore(reducers)}>
    {routes}
  </Provider>
), document.getElementById('app'))
```

Импортирует (объединенные) преобразователи из ./modules (./modules/index.js)

Применяет редьюсеры

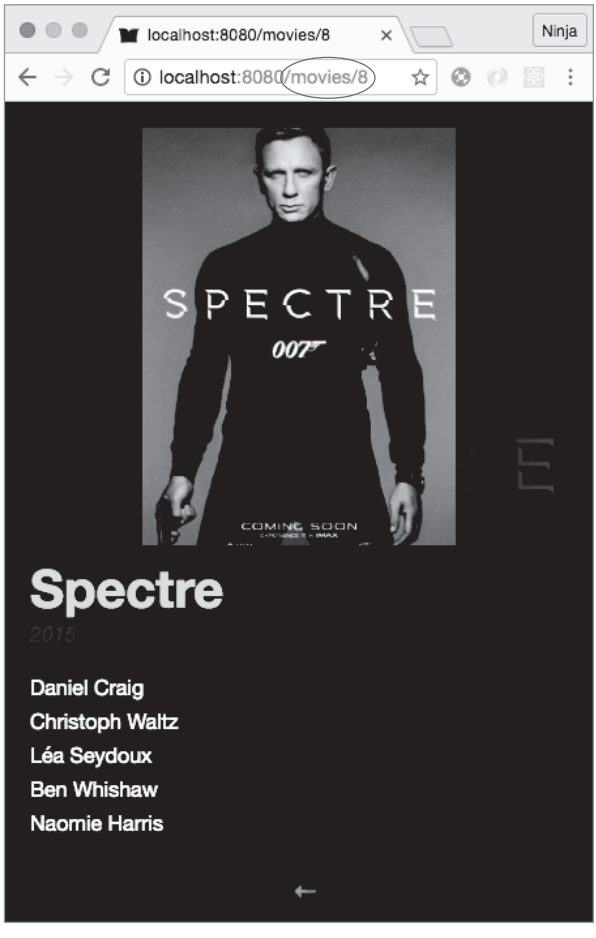


Рис. 14.9. Представление конкретного фильма на малом экране. URL-адрес включает идентификатор фильма

Что здесь происходит? Данные фильмов необходимо сохранить в хранилище. Возможно, в будущем в хранилище будут реализованы дополнительные части, например учетные записи пользователей или другие сущности. Мы воспользуемся функциональностью Redux, которая позволяет создать столько разных частей хранилища, сколько вам нужно, даже если в настоящее время нужна только одна. В некотором смысле средний шаг объединения редьюсеров способствует улучшению архитектуры, потому что позднее вы сможете легко расширить свое приложение, добавив новые редьюсеры в файл ./modules/index.js (или ./modules) с использованием паттерна Node[1]. Этот прием также называется *разделением редьюсеров* (http://mng.bz/Wprj).

[1] Azat Mardan, «Node Patterns: From Callbacks to Observer», webapplog, http://mng.bz/p9vd.

Каждый редьюсер может изменять данные в хранилище; но чтобы эта операция была безопасной, возможно, вам придется разделить состояние приложения на отдельные части, а затем объединить их в одно хранилище. Такой подход по принципу «разделяй и властвуй» рекомендуется для больших приложений с растущим количеством редьюсеров и действий. Для объединения нескольких редьюсеров удобно воспользоваться функцией `combineReducers()` из `redux` (ch14/redux-netflix/src/modules/index.js).

Листинг 14.5. Объединение редьюсеров

```
Применяет деструктурирующее присваивание ES6/ES2015 для создания объекта редьюсера с именем movies из свойства reducer в ./movies.js

Импортирует combineReducers из свойства combineReducers в redux

const { combineReducers } = require('redux')
const {
  reducer: movies
} = require('./movies')

module.exports = combineReducers({   <-- Экспортирует объединенный редьюсер movies
  movies
  // другие редьюсеры
})
```

Вы можете передавать сколько угодно редьюсеров и создавать независимые ветви в хранилище. Присваивайте им любые имена по своему усмотрению. В данном случае редьюсер `movies` импортируется, а затем передается функции `combineReducers()` как свойство простого объекта с ключом `"movies"`.

Тем самым вы объявляете отдельную часть хранилища и присваиваете ей имя «movies». Каждое действие, за которое отвечает редьюсер из `./movies`, затронет только эту часть и ничего более.

14.3.6. Редьюсер для movies

Теперь реализуем редьюсер «movies». *Редьюсер* в Redux представляет собой функцию, которая выполняется при каждой передаче любого события. Она выполняется с двумя аргументами:

- Первый аргумент `state` содержит ссылку на часть состояния, которой управляет данный редьюсер.
- Второй аргумент `action` содержит объект, представляющий действие, которое только что было передано при диспетчеризации[1].

[1] За подробной документацией по методу Array.prototype.reduce() обращайтесь к Mozilla Developer Network, http://mng.bz/Z55j.

Иначе говоря, входными данными редьюсера являются результаты предыдущих действий: текущее состояние (`state`) и текущее действие (`action`.) Редьюсер берет текущее состояние и применяет действие. Результатом работы редьюсера становится новое состояние. Если ваши редьюсеры представляют собой «чистые» функции без побочных эффектов (как это должно быть), вы сможете пользоваться всеми преимуществами использования Redux с React, такими как оперативная перезагрузка и возврат во времени.

РЕДЬЮСЕРЫ В JAVASCRIPT

Термин «редьюсер» происходит из функционального программирования. JavaScript отчасти обладает функциональной природой, поэтому в нем присутствует метод `Array.reduce()`.

В двух словах: метод `reduce` представляет собой операцию, которая обобщает список элементов — на входе несколько значений, на выходе только одно. Список, с которым работает редьюсер, может быть массивом, как в случае с JavaScript, или другой структурой данных, например списком, как в других языках.

Например, редьюсер может вернуть количество вхождений некоторого имени в списке имен. В этом случае входными данными является список имен, а выходными — количество вхождений.

Чтобы использовать редьюсер, вызовите метод и передайте функцию редьюсера, которая получает:

- *Аккумулятор* — то, что передается следующей итерации и со временем станет результатом вычислений;
- *Текущее значение* — элемент из списка.

С каждой итерацией по элементам списка (или массива в JS) функция редьюсера получает значение аккумулятора. В JavaScript метод называется `Array.reduce()`. Например, для получения частот имен в списке можно выполнить следующий код редьюсер, который использует тернарный оператор для проверки текущего значения (`curr`), и если оно равно `"azat"`, увеличивает аккумулятор (`acc`) на 1:

```
const users = ['azat', 'peter', 'wiliam','azat','azat']
console.log(users
  .reduce((acc, curr)=>(
    (curr == 'azat') ? ++acc : acc
  ), 0)
)
```

В редьюсерах Redux аккумулятором является объект состояния, а текущим значением — текущее действие. Результатом функции является новое состояние.

СОВЕТ Старайтесь избегать включения вызовов API в редьюсеры. Помните: редьюсеры должны быть «чистыми» функциями без побочных эффектов. По сути, они являются конечными автоматами, и они не должны выполнять асинхронные операции, такие как HTTP-вызовы к API. Подобные асинхронные вызовы лучше всего размещать в промежуточных модулях (http://redux.js.org/docs/advanced/Middleware.html) или в функции, которая создает действие `dispatch()` (http://mng.bz/S31I). Использование `dispatch()` в компоненте будет продемонстрировано позднее в этой главе.

Типичный редьюсер выглядит как функция с огромной конструкцией `switch/case`:

```
const FETCH_MOVIES = 'movies/FETCH_MOVIES'
const FETCH_MOVIE = 'movies/FETCH_MOVIE'

const initialState = {
  movies: [],
  movie: {}
}

function reducer(state = initialState, action) {
  switch(action.type) {
    case FETCH_MOVIES:
      return {
        ...state,   <-- Оператор расширения ES6 передает объект state в порядке ключей
        all: action.movies   <-- Сохраняет или изменяет список всех фильмов в хранилище
      }
    case FETCH_MOVIE:
      return {
        ...state,
        current: action.movie   <-- Сохраняет или изменяет конкретный фильм в хранилище
    }
  }
}

module.exports = {
  reducer   <-- Экспортирует объект с методом reducer с использованием синтаксиса ES6
}
```

Однако такой авторитет, как Дуглас Крокфорд (Douglas Crockford), в своей классической книге «JavaScript: The Good Parts» (O'Reilly Media, 2008) называет использование `switch/case` нежелательным. Существует удобная библиотека redux-actions (https://github.com/acdlite/redux-actions), которая позволяет представить функцию редьюсера в более элегантной, более функциональной форме. Вместо нагромождения `switch/case` используется более удобный объект.

Воспользуемся методом `handleActions` из `redux-actions`. Этот метод получает объект, сходный с ассоциативным массивом: ключами являются типы действий, а значениями — функции. Таким образом, для каждого типа действия вызывается только одна функция (иначе говоря, вызываемая функция определяется типом действия).

Функцию из предыдущего фрагмента можно переписать с `redux-actions` и `handleActions` так, как показано в листинге 14.6 (ch14/redux-netflix/src/modules/movies.js).

Листинг 14.6. Использование библиотеки redux-actions

```
const { handleActions } = require('redux-actions')

const FETCH_MOVIES = 'movies/FETCH_MOVIES'
const FETCH_MOVIE = 'movies/FETCH_MOVIE'

const initialState = {
  movies: [],
  movie: {}
}

module.exports = {
  fetchMoviesActionCreator: (movies) => ({
    type: FETCH_MOVIES,
    movies
  }),
  fetchMovieActionCreator: (index) => ({
    type: FETCH_MOVIE,
    index
  }),
  reducer: handleActions({
    [FETCH_MOVIES]: (state, action) => ({
      ...state,
      all: action.movies
    }),
    [FETCH_MOVIE]: (state, action) => ({
      ...state,
      current: state.all[action.index - 1]
    })
  }, initialState)
}
```

Определяет создателя действия FETCH_MOVIES, который возвращает объект действия

Определяет создателя действия FETCH_MOVIES, который возвращает объект действия

Получает все фильмы в компонент Movies

Получает текущий фильм в компонент Movie с использованием индекса (параметр URL с идентификатором фильма)

Этот код по своей структуре близок к реализации `switch/case`, но его суть заключается в отображении функций на действия, нежели в выборе их из потенциально огромной условной конструкции.

14.3.7. Действия

Для изменения данных в хранилище используются действия (actions). Уточню для ясности: действием может быть все что угодно, не только ввод данных пользователем в браузере. Например, действием может быть получение результата асинхронной операции. Фактически любой код может стать действием. Действия — единственные источники информации для хранилища; эти данные передаются из приложения в хранилище. Действия выполняются методом `store.dispatch()`,

о котором я упоминал ранее, или вспомогательным методом `connect()`. Но прежде чем разбираться с тем, как вызвать действие, рассмотрим его *тип*.

Каждое действие представляется простым объектом, у которого имеется как минимум одно свойство — `type`. Также он может содержать сколько угодно других свойств, которые обычно используются для передачи данных хранилищу. Таким образом, у любого действия имеется тип:

```
{
  type: 'movies/I_AM_A_VALID_ACTION'
}
```

Здесь действие имеет строковый тип.

> **ПРИМЕЧАНИЕ** Имена действий принято записывать символами верхнего регистра с префиксом из имени модуля, записанного в нижнем регистре. Если вы полностью уверены в отсутствии дубликатов, имя модуля можно опустить.

В современной разработке React типы действий объявляются в виде строковых констант:

```
const FETCH_MOVIES = 'movies/FETCH_MOVIES'
const FETCH_MOVIE = 'movies/FETCH_MOVIE'
```

В этом примере объявляются два типа действий. Оба типа представляют собой строки, состоящие из двух частей: имени модуля Redux и имени типа действия. Эта практика может быть полезной при наличии разных редьюсеров с одноименными действиями.

Каждый раз, когда вы захотите изменить состояние приложения, необходимо передать соответствующее действие. Будет выполнена соответствующая функция-редьюсер, что приведет к обновлению состояния приложения. Подумайте о данных, которые вы получаете от API или от формы, заполненной пользователем: все они могут быть использованы для размещения или обновления хранилища. Пример:

```
this.props.dispatch({
  type: FETCH_MOVIE,
  movie: {}
})
```

Последовательность действий выглядит так:

1. Вызвать `dispatch()` с объектом действия, который содержит свойство `type` и данные в компоненте (если они нужны).
2. Выполнить соответствующий редьюсер в модуле.
3. Обновить новое состояние в хранилище, доступ к которому осуществляется через компоненты.

Передача действий будет более подробно рассмотрена позднее. А сейчас посмотрим, как избежать передачи/использования типов действий в компонентах.

14.3.8. Создатели действий

Чтобы внести какие-либо изменения в хранилище, необходимо выполнить действие для всех редьюсеров. Редьюсер изменяет состояние приложения в зависимости от типа действия, поэтому *всегда необходимо знать тип действия*. Однако существует сокращенный синтаксис, позволяющий скрывать типы действий в создателях действий (action creators). Общая схема действий выглядит так:

1. Вызвать создателя действия с данными (если необходимо). Создатель действия может быть определен в модуле редьюсера.
2. Выполнить передачу действия в компоненте. *Тип действия для этого не нужен*.
3. Выполнить соответствующий редьюсер в модуле редьюсера.
4. Обновить новое состояние в хранилище.

Взгляните:

```
this.props.dispatch(fetchMoviesActionCreator({movie: {}}))
```

Создатель действия представляет собой функцию, которая возвращает действие. Реализация выглядит предельно прямолинейно:

```
function fetchMoviesActionCreator(movies) {
  return {
    type: FETCH_MOVIES,
    movies
  }
}
```

С создателями действий вы можете скрыть сложную логику в одном вызове функции. Впрочем, в данном случае никакой логики нет. Эта функция выполняет всего одну операцию — она возвращает действие: простой объект со свойством `type`, который определяет это действие, а также свойством `movies`, значением которого является массив с данными фильмов. Если вы захотите расширить клон Netflix и реализовать возможность добавления новых фильмов, вам потребуется создатель действия `addMovie()`:

```
function addMovie(movie) {
  return {
    type: ADD_MOVIE,
    movie
  }
}
```

А как насчет `watchMovie()`?

```
function watchMovie(movie, watchMovieIndex, rating) {
  return {
    type: WATCH_MOVIE,
    movie,
    index: watchMovieIndex,
    rating: rating,
    receivedAt: Date.now()
  }
}
```

Помните: действие *должно* иметь свойство `type`!

Чтобы иметь возможность передавать действия, необходимо связать компоненты с хранилищем. Ситуация становится более интересной, потому что мы подходим к обновлению состояния.

14.3.9. Установление связи компонентов с хранилищем

Теперь, когда вы научились помещать данные в хранилище, посмотрим, как обращаться к данным в хранилище из компонентов. К счастью, компонент `Provider` предоставляет функциональность для передачи данных компонентам в свойствах. Но чтобы обратиться к данным, необходимо явно установить связь вашего компонента с хранилищем.

По умолчанию компонент не связывается с хранилищем данных, а одного его нахождения где-то в иерархии компонента верхнего уровня `Provider` недостаточно. Почему? Потому что для некоторых компонентов установление связи требует явно выраженного согласия.

Если вы еще не забыли, в соответствии с рекомендациями React, существуют два типа компонентов — презентационные и контейнерные (см. главу 8). Презентационным компонентам хранилище не нужно; они только потребляют свойства. В то же время контейнерным компонентам необходимо хранилище и диспетчер. Даже в определении контейнерных компонентов в документации Redux говорится о том, что они подписываются на работу с хранилищем (http://mng.bz/p4f9). В сущности, `Provider` лишь автоматически предоставляет хранилище для всех компонентов, чтобы некоторые из них могли подписываться/устанавливать связь с ним. Таким образом, для контейнерных компонентов необходим как компонент `Provider`, так и хранилище.

Итак, компонент после установления связи может обращаться к любым данным из хранилища в своих свойствах. Как связать компоненты с хранилищем? Конечно же, методом `connect()`! Рассмотрим пример. Наш корневой компонент `App` использует компонент `Movies`, который как минимум должен содержать следующий код для вывода списка фильмов (реальный компонент `Movies` будет содержать больше кода):

```
class Movies extends React.Component {
  render() {
    const {
```

```
      movies = []
    } = this.props

    return (
      <div className={styles.movies}>
        {movies.map((movie, index) => (
          <div key={index}>
            {movie.title}
          </div>
        ))}
      </div>
    )
  }
}
```

В настоящее время компонент `Movies` не связан с хранилищем, хотя `Provider` и является его родителем. Чтобы установить связь, добавьте приведенный ниже фрагмент. Функция `connect()` входит в пакет `react-redux` и получает до четырех аргументов, но сейчас мы используем только один:

```
const { connect } = require('react-redux')
class Movies extends React.Component {
  ...
}
module.exports = connect()(Movies)
```

Функция `connect()` возвращает функцию, которая затем применяется к компоненту `Movies`. В результате вызова `connect()` с `Movies` и присутствия `Provider` в числе родителей компонент `Movies` связывается с хранилищем.

Теперь компонент `Movies` может получать любые данные из хранилища и передавать действия (неожиданно, правда?). Но чтобы получать данные в нужном формате, вы должны *перебрать состояния и передать их свойствам компонента*, создав простую функцию мэппинга (из-за необходимости возвращения результата правильнее было бы использовать термин «*выражение*»).

В некоторых учебниках встречается функция с именем `mapStateToProps()`, хотя она не обязана быть явно объявленной функцией. Использование анонимной «стрелочной» функции столь же элегантно и прямолинейно. Функция мэппинга выделяется в специальный метод `connect()` из `react-redux`. Напомню, что состояние передается в первом аргументе `connect()`:

```
module.exports = connect(function(state) {
  return state
})(Movies)
```

Также возможен более модный и эффектный стиль «неявного return» из ESNext, совместимый с React:

```
module.exports = connect(state => state)(Movies)
```

В такой конфигурации все состояние приложения берется из хранилища и помещается в свойства компонента `Movies`. Обычно в приложении используется только ограниченное подмножество этого состояния. В нашем примере `Movies` достаточно `movies.all`:

```
class Movies extends React.Component {
  render() {
    const {
      children,
      movies = [],
      params = {}
    } = this.props
    ...

module.exports = connect(({movies}) => ({
  movies: movies.all
}), {
  fetchMoviesActionCreator
})(Movies)
```

Ниже приведен фрагмент `Movie`, который отображает на состояние только `movies.current`:

```
class Movie extends React.Component {
  render() {
    const {
      movie = {
        starring: []
      }
    } = this.props
    ...

module.exports = connect(({movies}) => ({
  movie: movies.current
}), {
  fetchMovieActionCreator
})(Movie)
```

Если хранилище окажется пустым, компонент не получит никаких дополнительных свойств... потому что их нет.

Затем начинается волшебство Redux: каждый раз, когда обновляется часть хранилища, все компоненты, зависящие от этой части, получают новые свойства, а следовательно, рендерятся заново. Это происходит при передаче действия; таким образом, ваши компоненты теперь обладают слабыми связями и обновляются только при обновлении хранилища. Любой компонент может запустить это обновление, передавая соответствующее действие. Нет необходимости использовать классические функции обратного вызова, передаваемые в свойствах, и передавать их от компонента верхнего уровня до компонента с самым глубоким уровнем вложенности — просто свяжите свои компоненты с хранилищем.

14.3.10. Передача действий

Чтобы применить изменения к данным в хранилище, необходимо передать действие. После того как компонент будет связан с хранилищем, вы можете получать свойства, отображенные на свойства состояния приложения, а также получите в свое распоряжение свойство `dispatch`.

Метод `dispatch()` представляет собой функцию, которая получает действие в аргументе и передает его хранилищу. Таким образом, вы можете передать действие, вызывая `this.props.dispatch()` с действием:

```
componentWillMount() {
  this.props.dispatch({
    type: FETCH_MOVIE,
    movie: {}
  })
}
```

`type` — строковое значение, применяемое библиотекой Redux ко всем редьюсерам, соответствующим данному типу. После того как действие будет передано (что обычно означает изменение хранилища), все компоненты, которые связаны с хранилищем и обладают свойствами, отображенными из обновленной части состояния приложения, будут отрендерены заново. Вам не нужно проверять, какие компоненты следует обновить, и вообще что-то делать вручную. Вы можете использовать новые свойства в функции `render()` компонентов:

```
class Movie extends React.Component {
  render() {
    const {
      movie = {
        starring: []
      }
    } = this.props
  ...
```

Минимальное действие (объект с `type`) можно заменить создателем действия (функция `fetchMovieActionCreator()`):

```
const fetchMovieActionCreator = (response) => {
  type: FETCH_MOVIE,
  movie: response.data.data.movie
}
...
  componentWillMount() {
    ... // Запрос AJAX/XHR
    this.props.dispatch(fetchMovieActionCreator(response))
  }
```

Так как `fetchMovieActionCreator()` возвращает простой объект, идентичный объекту из предыдущего примера (`type` и ключи `movie`), вы можете вызвать функцию создателя действия `fetchMovieActionCreator()` и передать результат `dispatch()`.

1. Получить данные асинхронно (`response`).
2. Создать действие (`fetchMovieActionCreator()`).
3. Передать действие (`this.props.dispatch()`).
4. Инициировать редьюсер.
5. Обновить новое состояние в свойствах (`this.props.movie`).

14.3.11. Передача создателей действий в свойствах компонентов

Создатели действий могут определяться как функции прямо в файле компонента, но их можно использовать и другим способом: определить в модуле, импортировать и поместить в свойства компонента. Для этого можно воспользоваться вторым аргументом функции `connect()` и передать создатель действия как метод:

```
const {
  fetchMoviesActionCreator
} = require('modules/movies.js')
class Movies extends Component {
  ...
}
module.exports = connect(state => ({
  movies: state.movies.all
}),{
  fetchMoviesActionCreator
})(Movies)
```

Импортирует создатель действия из client/modules/movies.js

Распределяет данные для заполнения свойства movies

Теперь вы можете обратиться к `fetchMovieActionCreator()` через свойства и передать действие без использования `dispatch()`:

```
class Movies extends Component {
  componentWillMount() {
    this.props.fetchMoviesActionCreator()
  }
  render() {
    const {
      movies = []
    } = this.props

    return (
      <div className={styles.movies}>
        {movies.map((movie, index) => (
          <div key={index}>
            {movie.title}
          </div>
        ))}
      </div>
    )
  }
}
```

Вызывает создатель действия напрямую для диспетчеризации действия

Присваивает movies значение this.props.movies или пустой массив (деструктуризация ES6)

Новый создатель действия автоматически упаковывается в действительный вызов `dispatch()`. Вам не придется заниматься этим самостоятельно. Потрясающе! Именно так компонент `Movies` реализуется в файле ch14/redux-netflix/src/components/movies/movies.js.

Для ясности можно переименовать `fetchMoviesActionCreator()` в `fetchMovies()` или поступить так:

```
const {
  fetchMoviesActionCreator
} = require('modules/movies.js')
class Movies extends Component {
  componentWillMount() {
    this.props.fetchMovies()  ◄——  Передает вызовом fetchMovies()
  }
  ...
module.exports = connect(state => ({
  movies: state.movies.all
}), {
  fetchMovies: fetchMoviesActionCreator  ◄——  Переименовывает метод действия
})(Movies)
```

Первый аргумент `connect()` — функция, отображающая состояние на свойства компонента, — получает все состояние (`state`) в единственном аргументе и возвращает простой объект с одним свойством `movies`:

```
...
module.exports = connect(state => ({
  movies: state.movies.all
}), {
  fetchMoviesActionCreator
})(Movies)
```

Деструктуризация `state.movies` делает код более выразительным:

```
module.exports = connect(({movies}) => ({
  movies: movies.all
}), {
  fetchMoviesActionCreator
})(Movies)
```

В функции `render()` компонента `Movies` значение `movies` берется из свойств и рендерится в коллекции одноранговых элементов DOM. Каждый из них представляет собой элемент `div`, у которого внутреннему тексту присваивается `movie.title` — типичный способ рендеринга массива во фрагменте с одноранговыми элементами DOM.

Интересуетесь, как будет выглядеть итоговая версия компонента `Movies`? Код приведен в листинге 14.7 (ch14/redux-netflix/src/components/movies/movies.js).

Листинг 14.7. Передача создателей действий в свойствах Movies

```
const React = require('react')
const { connect } = require('react-redux')
const { Link } = require('react-router')
const movies = require('../../movies.json')
const {
  fetchMoviesActionCreator
} = require('modules/movies.js')
const styles = require('./movies.css')

class Movies extends React.Component {
  componentWillMount() {
    this.props.fetchMovies(movies)
  }

  render() {
    const {
      children,
      movies = [],
      params = {}
    } = this.props

    return (
      <div className={styles.movies}>
        <div className={params.id ? styles.listHidden : styles.list}>
          {movies.map((movie, index) => (
            <Link
              key={index}
              to={`/movies/${index + 1}`}>
              <div
                className={styles.movie}
                style={{backgroundImage: `url(${movie.cover})`}} />
            </Link>
          ))}
        </div>
        {children}
      </div>
    )
  }
}

module.exports = connect(({movies}) => ({
  movies: movies.all
}), {
  fetchMovies: fetchMoviesActionCreator
})(Movies)
```

Загружает фиктивную базу данных из файла JSON (с использованием json-loader) в movies

Передает действие с использованием fetchMoviesActionCreator() (FETCH_MOVIES) с данными из объекта JSON movies, что может быть заменено вызовом AJAX/XHR к API-серверу

Передает дочерние элементы в соответствии с определением в иерархии React Router

Связывает компонент с хранилищем, в результате чего он получает доступ к данным и создателю действия fetchMoviesActionCreator() из свойств

Как видите, переход на асинхронные данные происходит достаточно просто: выполните асинхронный вызов (с использованием `fetch()` API, axios и т. д.), а затем передайте действие в `componentWillMount()`. Или еще лучше, используйте метод `componentDidMount()`, как рекомендует команда React для вызовов AJAX/XHR:

```
componentWillMount() {
  // this.props.fetchMovies(movies)
}

componentDidMount() {
  fetch('/src/movies.json', {method: 'GET'})
    .then((response)=>{return response.json()})
    .then((movies)=>{
      this.props.fetchMovies(movies)
    })
}
```

Не передает действие с данными, импортированными require (синхронно)

Получает файл JSON, который будет предоставляться WDS (асинхронно)

Передает действие с данными, которые были получены асинхронно от сервера запросом GET

То же, что было сделано с GET, можно сделать с POST, PUT и другими командами HTTP. Некоторые из этих команд будут использованы в следующей главе.

С `Movies` мы закончили. Далее мы рассмотрим компонент `Movie` — но только кратко, потому что большая часть кода Redux очень похожа на код `Movies`. Отличается прежде всего тем, что `Movie` получает из параметра URL идентификатор фильма. React Router помещает его в `this.props.params.id`. Этот идентификатор будет передаваться и использоваться для выбора только одного фильма. Напомню, как выглядят редьюсеры из src/modules/movies.js:

```
...
  reducer: handleActions({
    [FETCH_MOVIES]: (state, action) => ({
      ...state,
      all: action.movies
    }),
    [FETCH_MOVIE]: (state, action) => ({
      ...state,
      current: state.all[action.index - 1]
    })
  },
...
```

Использует индекс для выборки одного фильма

А теперь рассмотрим реализацию компонента `Movie`, которая использует другой мэппинг состояния на свойства, для чего компонент берет идентификатор фильма из параметра URL React Router и использует его как индекс (src/components/movie/movie.js).

Листинг 14.8. Реализация Movie

```
const React = require('react')
const { connect } = require('react-redux')
const { Link}=require('react-router')
const {
  fetchMovieActionCreator
} = require('modules/movies.js')
const styles = require('./movie.css')
```

Импортирует файл CSS

```
class Movie extends React.Component {
  componentWillMount() {
    this.props.fetchMovie(this.props.params.id)
  }
  componentWillUpdate(next) {
    if (this.props.params.id !== next.params.id) {
      this.props.fetchMovie(next.params.id)
    }
  }
  render() {
    const {
      movie = {
        starring: []
      }
    } = this.props

    return (
      <div
        className={styles.movie}
        style={{backgroundImage: `linear-gradient(90deg, rgba(0, 0, 0, 1) 0%,
         ➥ rgba(0, 0, 0, 0.625) 100%), url(${movie.cover})`}}>
        <div
          className={styles.cover}
          style={{backgroundImage: `url(${movie.cover})`}} />
        <div className={styles.description}>
          <div className={styles.title}>{movie.title}</div>
          <div className={styles.year}>{movie.year}</div>
          <div className={styles.starring}>
            {movie.starring.map((actor = {}, index) => (
              <div
                key={index}
                className={styles.actor}>
                {actor.name}
              </div>
            ))}
          </div>
        </div>
        <Link
          className={styles.closeButton}
          to="/movies">
          ←
        </Link>
      </div>
    )
  }
}

module.exports = connect(({movies}) => ({
  movie: movies.current
}), {
  fetchMovie: fetchMovieActionCreator
})(Movie)
```

Передает только при изменении параметра URL

Применяет стили к элементам

Отображает данные от преобразователя на свойство

14.3.12. Выполнение клона Netflix

Пора запустить проект. Конечно, это можно было сделать с самого начала, потому что сценарий запуска находится в файле package.json. Этот сценарий использует библиотеку npm concurrently для одновременного выполнения двух процессов: сборки Webpack в режиме отслеживания изменений и сервера разработки Webpack (порт 8080):

```
"start": "concurrently \"webpack --watch --config webpack.config.js\"
➥ \"webpack-dev-server\""
```

Перейдите в корневой каталог проекта (ch14/redux-netflix). Установите зависимости командой `npm i` и запустите проект из папки проекта командой `npm start`. Откройте свой любимый браузер на странице http://localhost:8080.

Поэкспериментируйте и убедитесь в том, что маршрутизация работает, а изображения загружаются независимо от того, используются ли фиктивные данные (`require()`) или информация загружается запросом GET. Обратите внимание: если вы находитесь на странице http://localhost:8080/movies/1 и обновите страницу, вы ничего не увидите. Эта проблема будет решена в следующей главе, где мы реализуем Node и сервер Express для поддержки URL без идентификаторов фрагментов. А эту главу пора заканчивать.

14.3.13. Redux: заключение

Redux предоставляет единое место для хранения всех данных приложения, и изменять эти данные можно только одним способом — посредством действий. Тем самым обеспечивается универсальность Redux — его можно использовать где угодно, не только в приложениях React. Но в библиотеке `react-redux` функция `connect()` позволяет связать любой компонент с хранилищем и заставить его реагировать на любые изменения.

Так выглядит базовая идея реактивного программирования: сущность A, отслеживающая изменения в сущности B, реагирует на эти изменения при их возникновении — но не наоборот. Здесь A — любой из ваших компонентов, а B — хранилище.

После установления связи компонента с хранилищем (`connect()`) и отображения свойств хранилища на свойства компонента (`this.props`) вы можете обращаться к последним в функции `render()`. Обычно для обращения к данным необходимо сначала обновить хранилище этими данными. Вот почему действие вызывается в функции `componentWillMount()` компонента. В тот момент, когда произойдет исходное подключение компонента и будет сделан вызов `render()`, часть хранилища, к которой обращается компонент, может быть пустой. Но после обновления данных хранилища эти изменения сохраняются. Вот почему в клоне Netflix список фильмов остается без изменений даже после навигации в приложении (по страницам или представлениям). Да, данные не исчезают из хранилища после отключения

компонента, в отличие от использования состояния компонента (помните `this.state()` и `this.setState()`?). Таким образом, хранилище Redux может обслуживать разные части вашего приложения, требующие тех же данных без необходимости перезагрузки данных.

Также безопасно выполняется обновление свойств компонентов в функции `render()` через хранилище с передачей действия, так как эта операция откладывается. С другой стороны, *без* Redux вы не сможете использовать `setState()` в любой момент, когда может потребоваться обновление компонента: в `render()`, `componentWillMount()` или `componentWillUpdate()`. Эта особенность технологии Redux — одна из составляющих ее гибкости.

14.4. Вопросы

1. Назовите два главных аргумента функции редьюсера, передаваемой методу `Array.reduce()` в JavaScript.
2. Redux предоставляет простоту, большую экосистему и удобство для разработчика по сравнению с Facebook Flux (`flux`). Да или нет?
3. Какую из следующих конструкций вы бы использовали для создания хранилища и провайдера: `new Provider (createStore(reducers))`, `<Provider store={createStore(reducers)}>` или `provider(createStore(reducers))`.
4. Redux необходим диспетчер, потому что так определено в Flux. Да или нет?
5. В этом проекте `movies.all` получает все фильмы, а `movies.current` получает текущий фильм. Они используются в компонентах `Movies` и `Movie` соответственно, в вызове `connect`. Где бы вы определили логику `movies.all` и `movies.current`?

14.5. Итоги

- Однонаправленная передача данных обеспечивает предсказуемое поведение и простоту сопровождения для приложений React.
- Для работы с React и однонаправленной передачи данных рекомендуется использовать архитектуру Flux.
- Redux — одна из самых популярных реализаций архитектуры Flux.
- В Redux возможна передача действия или помещение его в объект свойств.
- Функция Redux `connect()` позволяет обращаться к данным хранилища и выполнять передачу действий — обе эти возможности необходимы для контейнерных компонентов.

- Компонент Redux `Provider` предоставляет дочерним компонентам доступ к хранилищу, чтобы хранилище не приходилось вручную передавать в свойствах.
- Редьюсер представляет собой файл, содержащий функцию преобразования, которая (обычно) использует команду `switch/case` или `handleActions` для применения действий к новому состоянию: на вход подаются текущее состояние и действия, а на выходе получается новое состояние.
- Функция Redux `combineReducers` объединяет несколько редьюсеров, позволяя разделить код этих редьюсеров по нескольким файлам/модулям.

14.6. Ответы

1. Два главных аргумента — аккумулятор и текущее значение. Без них обработка списка невозможна.
2. Да. См. вводную часть этой главы и пост Дэна Абрамова (Dan Abramov) «Why Use Redux over Facebook Flux?» на сайте Stack Overflow (http://mng.bz/z9ok).
3. `<Provider store={createStore(reducers)}>`.
4. Нет. Redux реализует Flux, но не требует обязательного присутствия диспетчера, поэтому Redux реализуется проще.
5. В src/modules/movies.js.

15 Работа с данными в GraphQL

Посмотрите вступительный видеоролик к этой главе, отсканировав QR-код или перейдя по ссылке http://reactquickly.co/videos/ch15.

ЭТА ГЛАВА ОХВАТЫВАЕТ СЛЕДУЮЩИЕ ТЕМЫ:

- Запрос данных с сервера с помощью GraphQL и Axios.
- Поставка данных в хранилище Redux.
- Реализация бэкенда GraphQL с помощью Node/Express.
- Поддержка маршрутизации URL без хеша.

В главе 14 мы реализовали клон Netflix на базе Redux. Данные были взяты из файла JSON, но вы также могли использовать REST-совместимый вызов API с использованием `axios` или `fetch()`. В этой главе рассматривается один из самых популярных вариантов представления данных в интерфейсе приложения: GraphQL.

В предыдущей главе мы импортировали файл JSON как внутреннее хранилище данных или использовали REST-совместимые вызовы для получения того же файла с эмуляцией REST-совместимой конечной точки GET. Имитация API хорошо подходит для создания прототипов, потому что клиентская часть уже готова; когда вам потребуется долгосрочное хранение данных, имитации заменяются сервером, который обычно предоставляет REST API (или, при *крайней* необходимости, — SOAP[1]).

Представьте, что API клона Netflix должен разрабатываться другой командой. Вы согласовываете формат данных JSON (или XML) в ходе нескольких встреч.

[1] SOAP — в основном устаревший протокол, который в значительной мере основан на XML, а в последнее время был заменен REST.

Другая команда выдает результат. Все нормально стыкуется, и ваше приложение клиентской части получает все данные. Потом владельцы продукта общаются с заказчиками и решают, что им нужно новое поле для показа рейтинга фильмов. Что произойдет дальше? Вы должны реализовать новую конечную точку `movies/:id/ratings`, или другой команде придется перерабатывать старую версию и добавлять новое поле.

А может быть, приложение все еще находится в фазе построения прототипа. В таком случае поле, вероятно, удастся добавить в существующую страницу `movies/:id`. Нетрудно представить, что со временем вы будете получать и другие запросы на изменение форматов и структур. А если рейтинги тоже должны выводиться в `movies`? А если понадобятся новые вложенные поля из других коллекций — скажем, рекомендации друзей? В наш век быстрой гибкой разработки и бережливой (lean) методологии гибкость является важным преимуществом. Чем быстрее эти поля и данные могут адаптироваться к конечному продукту, тем лучше. Элегантное решение GraphQL избавляет разработчика от многих проблем такого рода.

> **ПРИМЕЧАНИЕ** Исходный код примеров этой главы доступен по адресам www.manning.com/books/react-quickly и https://github.com/azat-co/react-quickly/tree/master/ch15. Также некоторые демонстрационные примеры доступны по адресу http://reactquickly.co/demos.

15.1. GraphQL

В этой главе мы продолжим разработку клона Netflix и добавим к нему сервер. Этот сервер будет предоставлять GraphQL API — современный способ организации доступа к данным в приложениях React. GraphQL часто используется в сочетании с Relay; но как показывает этот пример, GraphQL также можно использовать с Redux или любой другой браузерной библиотекой данных. Для запросов AJAX/XHR/HTTP будет использоваться `axios`.

Когда вы работаете с GraphQL и Redux, сервер (серверная часть и веб-сервер) может быть написан на чем угодно (Ruby, Python, Java, Go, Perl), не обязательно на Node.js, но я рекомендую использовать Node, и именно этот вариант будет рассмотрен в этом разделе, потому что позволяет использовать JavaScript во всем технологическом стеке разработки.

Вкратце GraphQL (https://github.com/graphql/graphql-js) использует строки запросов, которые интерпретируются сервером (обычно Node), который, в свою очередь, возвращает данные в формате, заданном этими запросами. Запросы пишутся в JSON-подобном формате:

```
{
  user(id: 734903) {
    id,
    name,
```

```
    isViewerFriend,
    profilePicture(size: 50) {
      uri,
      width,
      height
    }
  }
}
```

А ответ содержит старый добрый JSON:

```
{
  "user" : {
    "id": 734903,
    "name": "Michael Jackson",
    "isViewerFriend": true,
    "profilePicture": {
      "uri": "https://twitter.com/mjackson",
      "width": 50,
      "height": 50
    }
  }
}
```

Сервер клона Netflix может использовать REST или более старые стандарты SOAP. Однако с новым паттерном GraphQL можно использовать инверсию управления: клиенты (приложения клиентской части или мобильные приложения) сами решают, какие данные им нужны, — вместо того чтобы программировать эту логику в серверных конечных точках/маршрутах. Некоторые преимущества инверсии управления[1]:

- Запросы, определяемые клиентом, — клиент получает в точности ту информацию, которая ему необходима.
- Структура, произвольный код — унифицированный API обеспечивает необходимую гибкость на стороне сервера.
- Сильная типизация — более надежная проверка данных и большая определенность в ответах, а также упрощенное использование данных в языках с сильной типизацией, таких как TypeScript, Swift, Java и Objective-C.
- Иерархические запросы — запросы соответствуют возвращаемым данным, а это важно, поскольку данные используются иерархическими представлениями.
- Ускорение построения прототипа — нет необходимости в обширной разработке серверной части или в больших раздельно работающих командах серверной части и API, потому что запрос имеет одну конечную точку.

[1] За дополнительной информацией о преимуществах GraphQL, в том числе и о сильной типизации, обращайтесь к статье Ника Шрока (Nick Schrock) «GraphQL Introduction», React, 1 мая 2015 г., http://mng.bz/DS65.

- Сокращение количества вызовов API — приложение клиентской части выдает меньше запросов к серверу, потому что структура данных определяется приложением клиентской части и может содержать информацию, которая ранее могла быть получена только из нескольких конечных точек REST.

RELAY И RELAY MODERN

GraphQL API можно использовать в приложениях React при помощи Relay (https://facebook.github.io/relay; graphql-relay-js и react-relay в npm). Некоторые разработчики предпочитают использовать Relay вместо Redux при работе с серверной частью GraphQL. Рассмотрев примеры, приведенные в документации, вы заметите сходство со связыванием компонентов в Redux; вместо хранилища используется удаленный GraphQL API.

Если React позволяет определять представления в виде компонентов и строить сложные пользовательские интерфейсы и приложения из множества простых компонентов, Relay позволяет компонентам указать, какие данные им нужны; это способствует локализации требований к данным. Компоненты React не зависят от логики и рендера других компонентов.

То же можно сказать и о Relay: компоненты держат свои данные ближе к себе, что упрощает композицию (построение сложных пользовательских интерфейсов и приложений из простых компонентов). Relay Modern — самая новая версия Relay. Она проще в использовании и предоставляет больше возможностей для расширения[1]. Если вы или ваша команда собираетесь серьезно использовать GraphQL, я настоятельно рекомендую также обратить внимание на Relay/Relay Modern.

15.2. Добавление сервера к клону Netflix

Чтобы доставить данные в приложение React, мы воспользуемся простым сервером на базе Express (https://github.com/expressjs/express) и GraphQL. Express хорошо подходит для упорядочения и предоставления доступа к конечным точкам API, а GraphQL обеспечивает доступность данных в формате, удобном для браузера (JSON).

Структура проекта выглядит так (большая часть кода взята из `redux-netflix`):

```
/redux-graphql-netflix
  /build  <-- Откомпилированные файлы
    /public  <-- Откомпилированные файлы клиентской части
      index.js
      style.css
    server.js  <-- Откомпилированные файлы серверной части
  /client  <-- Файлы с исходным кодом React для клиентской части
```

[1] См. https://facebook.github.io/relay/docs/relay-modern.html.

```
/components
  /app
    app.css
    app.js
  /movie
    movie.css
    movie.js
  /movies
    movies.css
    movies.js
  /modules
    index.js
    movies.js
  index.js
  routes.js
/node_modules
/server
  index.js   <-- Файл с исходным кодом Express для серверной части
  movies.json
  schema.js  <-- Схема GraphQL
index.html
package.json
webpack.config.js
webpack.server.config.js
```

Данные и на этот раз будут загружаться из файла JSON, но теперь файл находится на сервере. Вы можете легко заменить файл JSON movies.json обращениями к базе данных в server/schema.js. Но прежде чем рассматривать схемы, установим все необходимые зависимости, включая Express.

В листинге 15.1 приведен файл package.json (ch15/redux-graphql-netflix/package.json). Знаете, что делать? Конечно же, скопировать его и выполнить команду `npm i`!

Листинг 15.1. Файл package.json для клона Netflix

```
{                                          Добавляет сценарий запуска, который откомпилирует
  "name": "redux-graphql-netflix",         серверный и браузерный код и запустит сервер
  "version": "1.0.0",
  "description": "A sample project in React, GraphQL, Express and Redux that
  ➥ copies Netflix's features and workflow",
  "main": "index.js",
  "scripts": {
    "start": "concurrently \"webpack --watch --config webpack.config.js\"
    ➥ \"webpack --watch --config
    ➥ webpack.server.config.js\" \"webpack-dev-server\" \"nodemon
    ➥ ./build/server.js\""
  },
  "repository": {
    "type": "git",
    "url": "git+https://github.com/azat-co/react-quickly.git"
  },
```

```
  "author": "Azat Mardan (http://azat.co)",
  "license": "MIT",
  "bugs": {
    "url": "https://github.com/azat-co/react-quickly/issues"
  },
  "homepage": "https://github.com/azat-co/react-quickly#readme",
  "devDependencies": {
    "babel-core": "6.11.4",
    "babel-eslint": "6.1.2",
    "babel-loader": "6.2.4",
    "babel-polyfill": "6.9.1",
    "babel-preset-es2015": "6.9.0",
    "babel-preset-react": "6.11.1",
    "babel-preset-stage-0": "6.5.0",
    "concurrently": "2.2.0",
    "css-loader": "0.23.1",
    "eslint": "3.1.1",
    "eslint-plugin-babel": "3.3.0",
    "eslint-plugin-react": "5.2.2",
    "extract-text-webpack-plugin": "1.0.1",
    "json-loader": "0.5.4",
    "nodemon": "1.10.0",
    "style-loader": "0.13.1",
    "webpack": "1.13.1",
    "webpack-dev-server": "1.14.1",
    "axios": "0.13.1",
    "clean-tagged-string": "0.0.1-b6",
    "react": "15.2.1",
    "react-dom": "15.2.1",
    "react-redux": "4.4.5",
    "react-router": "2.6.0",
    "redux": "3.5.2",
    "redux-actions": "0.10.1"
  },
  "dependencies": {
    "express": "4.14.0",
    "express-graphql": "0.5.3",
    "graphql": "0.6.2"
  }
}
```

Добавляет инструмент разработки nodemon для запуска и перезапуска Express

Добавляет axios для выдачи вызовов API с обещаниями (по аналогии с fetch), используемыми в клиентской части

Добавляет служебную программу для удаления пробелов из строковых шаблонов ES6 и т. д.

Добавляет фреймворк веб-сервера Express Node для использования в серверной части

Добавляет плагин GraphQL для Express, который будет использоваться как в клиентской, так и в серверной части

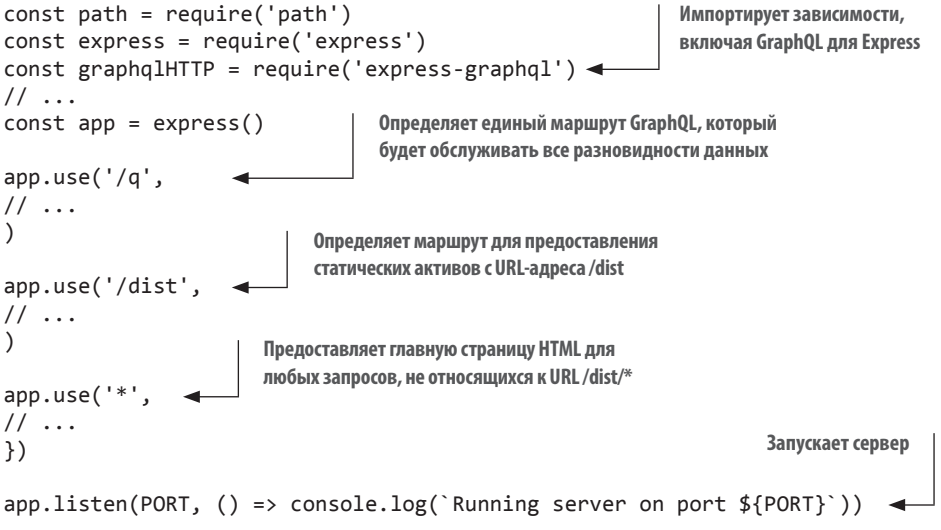
Добавляет GraphQL для использования как в серверной, так и в клиентской части

Перейдем к реализации основного серверного файла server/index.js.

15.2.1. Установка GraphQL на сервере

Основной рабочей частью веб-сервера, реализованного с Express и Node, является его начальная точка (иногда называемая точкой входа) — index.js. Этот файл находится в папке server, потому что он используется только в серверной части и не

должен быть доступным для клиентов по соображениям безопасности (он может содержать ключи API и пароли). Высокоуровневая структура файла выглядит так:



А теперь заполним пропуски. Для начала следует помнить, что один и тот же файл index.html должен поставляться для каждого маршрута, кроме конечной точки API и пакетных файлов. Это необходимо, потому что при использовании API-истории HTML5 и переходе к URL без идентификатора фрагмента (например, `/movies/8`) обновление страницы заставит браузер запросить тот же адрес.

Вероятно, вы заметили, что предыдущая версия клона Netflix при обновлении/перезагрузке страницы отдельного фильма (например, `/movies/8`) ничего не выводила. Дело в том, что для работы истории браузера необходимо еще кое-что реализовать. Этот код должен находиться на сервере, и он отвечает за отправку главного файла index.html для всех запросов (даже `/movies/8/`).

В Express при объявлении единой операции для каждого маршрута можно использовать символ * (звездочка):

```
app.use('*', (req, res) => {
  res.sendFile('index.html', {
    root: PWD
  })
})
```

В качестве альтернативы я рекомендую использовать промежуточный код Express с именем `express.static()`:

```
app.use('/dist',
  express.static(path.resolve(PWD, 'build', 'public'))
)
```

СОВЕТ За дополнительной информацией о промежуточном ПО и рекомендациями по использованию Express обращайтесь к приложению В и моим книгам «Pro Express.js» (Apress, 2014) и «Express Deep API Reference» (Apress, 2014).

STATIC, PUBLIC И DIST

Важность присутствия каталога public в сборке *невозможно* переоценить. Если не ограничить акт предоставления ресурсов (скажем, файлов) подкаталогом (например, `dist` или `public`), то весь ваш код будет доступен любому посетителю сервера. Даже такой внутренний код, как server.js, может быть доступен извне, если вы забудете использовать подкаталог. Например, следующий фрагмент:

```
// Антипаттерн. Не делайте так, а то вас уволят!
app.use('/dist',
  express.static(path.resolve(PWD, 'build'))
)
```

откроет доступ к файлу server.js для атакующих, а этот файл может содержать секреты, ключи API, пароли и подробности реализации по URL `/dist/server.js`.

Используя подкаталог (такой, как dist или public), открывая внешний доступ (через HTTP) только к нему и размещая только файлы клиентской части в открытом подкаталоге, можно ограничить несанкционированный доступ к другим файлам.

Для работы GraphQL API необходимо создать еще один маршрут (`/q`), в котором библиотека `graphqlHTTP` используется со схемой (server/schema.js) и сеансом (`req.session`) для возвращения данных:

```
app.use('/q', graphqlHTTP(req => ({
  schema,
  context: req.session
})))
```

И наконец, чтобы сервер заработал, необходимо приказать ему прослушивать входящие запросы на определенном порте:

```
app.listen(PORT, () => console.log(`Running server on port ${PORT}`))
```

Здесь `PORT` — переменная среды. Эта переменная может передаваться процессу из интерфейса командной строки:

```
PORT=3000 node ./build/server.js
```

NODEMON И NODE

Напомню, что в package.json мы использовали nodemon:

```
nodemon ./build/server.js
```

Команда nodemon делает то же, что и node, но перезапускает код при внесении изменений в него.

ПРЕДУПРЕЖДЕНИЕ В главе 14 используется порт 8080, потому что это значение используется по умолчанию Webpack Development Server. Нет абсолютно ничего плохого в использовании 8080 с сервером Express в данном примере, но по каким-то загадочным историческим причинам принято запускать приложения Express на порте 3000. Возможно, в этом следует винить Rails!

Сервер также использует другую переменную, записанную в верхнем регистре, — `PWD`. Это тоже переменная среды, но она задается Node и содержит каталог проекта: путь к папке, в которой находится файл package.json (то есть к корневой папке проекта).

И наконец, мы используем переменные `graphqlHTTP` и `schema`. Переменная `graphqlHTTP` получается из пакета `express-graphql`, а `schema` содержит вашу схему данных, построенную с использованием определений GraphQL.

В листинге 15.2 показана полная настройка сервера (ch15/redux-graphql-netflix/server/index.js).

Листинг 15.2. Сервер Express для предоставления данных и статических ассетов

```
const path = require('path')
const express = require('express')
const graphqlHTTP = require('express-graphql')

const schema = require('./schema')
const {
  PORT = 3000,
  PWD = __dirname
} = process.env
const app = express()

app.use('/q', graphqlHTTP(req => ({
  schema,
  context: req.session
})))

app.use('/dist', express.static(path.resolve(PWD, 'build', 'public')))

app.use('*', (req, res) => {
  res.sendFile('index.html', {
    root: PWD
  })
})

app.listen(PORT, () =>,
  console.log(`Running server on port ${PORT}`))
```

Сохраняет рабочий каталог этого файла (PWD = Print Working Directory, то есть «Вывод рабочего каталога»)

Запускает сервер с портом 3000 (вместо 8080)

GraphQL обладает сильной типизацией; это означает использование схем, как упоминалось ранее для `/q`. Схема определяется в server/schema.js, как было показано в описании структуры проекта. Давайте посмотрим, как выглядят данные, ведь схема определяется структурой данных.

15.2.2. Структура данных

В приложении используется пользовательский интерфейс для отображения информации о фильмах. Следовательно, эти данные должны где-то храниться. Проще всего сохранить их в файле JSON (server/movies.json). Файл содержит все фильмы, и каждый фильм представляется простым объектом с набором свойств, так что весь файл представляет собой массив объектов:

```
[{
  "title": "Pirates of the Caribbean: On Stranger Tides"
  ...
}, {
  "title": "Pirates of the Caribbean: At World's End"
  ...
}, {
  "title": "Avengers: Age of Ultron"
  ...
}, {
  "title": "John Carter"
  ...
}, {
  "title": "Tangled"
  ...
}, {
  "title": "Spider-Man 3"
  ...
}, {
  "title": "Harry Potter and the Half-Blood Prince"
  ...
}, {
  "title": "Spectre"
  ...
}, {
  "title": "Avatar"
  ...
}, {
  "title": "The Dark Knight Rises"
  ...
}]
```

> **ПРИМЕЧАНИЕ** В этом примере использованы данные 10 самых высокобюджетных фильмов из Википедии (https://en.wikipedia.org/wiki/List_of_most_expensive_films).

Каждый объект содержит различную информацию: название фильма, URL-адрес обложки, год выпуска, затраты на съемку в миллионах долларов и ведущие актеры. Например, объект «Пираты Карибского моря» содержит следующие данные:

```
{
  "title": "Pirates of the Caribbean: On Stranger Tides",
  "cover": "/dist/images/On_Stranger_Tides_Poster.jpg",
```

```
  "year": "2011",
  "cost": 378.5,
  "starring": [{
    "name": "Johnny Depp"
  }, {
    "name": "Penélope Cruz"
  }, {
    "name": "Ian McShane"
  }, {
    "name": "Kevin R. McNally"
  }, {
    "name": "Geoffrey Rush"
  }]
}
```

В текущем варианте каждый фильм представлен объектом, который содержит только название. Позднее вы можете добавить столько свойств, сколько потребуется; а пока сосредоточимся на схеме данных.

15.2.3. Схема GraphQL

С GraphQL можно использовать любой источник данных: базу данных SQL, хранилище объектов, группу файлов или удаленный API. Важны только два обстоятельства:

- «Чистота» данных — то есть для идентичных запросов должны возвращаться идентичные ответы (идемпотентность).
- Возможность представления данных в JSON.

У нас имеется список фильмов, хранящийся в файле JSON. Импортируем его в приложение:

```
const movies = require('./movies.json')
```

Типичная схема GraphQL определяет запрос с полями и аргументами. Схема данных в примере содержит только список объектов, и каждый объект обладает единственным свойством `title`. Определение схемы приводится ниже. Это очень простой пример — полная схема будет приведена позднее:

```
const movies = require('./movies.json')
new graphql.GraphQLSchema({
  query: new graphql.GraphQLObjectType({
    name: 'Query',
    fields: {
      movies: {
        type: new graphql.GraphQLList(new graphql.GraphQLObjectType({
          name: 'Movie',
          fields: {
            title: {
```

Импортирует фильмы из файла (имитация базы данных)

Определяет в схеме строковое поле title

```
            type: graphql.GraphQLString
          }
        }
      })),
      resolve: () => movies
    }
  }
})
})
```

Определяет get-метод для этого запроса, который будет отправлять данные из файла JSON (мог бы быть вызов базы данных)

Основная идея заключается в том, что при выполнении запроса выполняется функция, присвоенная ключу `resolve`. После этого из результата вызова функции выбираются только запрошенные свойства объектов. Эти свойства присутствуют в итоговых объектах, а поля, которые не были указаны, туда не попадут. Следовательно, вы должны указать, какие свойства вы хотите получить, при каждом выполнении запроса. Таким образом обеспечивается гибкость и эффективность вашего API: вы можете скомпоновать фрагменты данных так, как считаете нужным, на стадии выполнения.

В примере используются два типа запросов и другие поля. В листинге 15.3 приведена возможная реализация (ch15/redux-graphql-netflix/server/schema.js).

Листинг 15.3. Схема GraphQL

```
const {
  GraphQLSchema,
  GraphQLObjectType,
  GraphQLList,
  GraphQLString,
  GraphQLInt,
  GraphQLFloat
} = require('graphql')
const movies = require('./movies.json')

const movie = new GraphQLObjectType({
  name: 'Movie',
  fields: {
    title: {
      type: GraphQLString
    },
    cover: {
      type: GraphQLString
    },
    year: {
      type: GraphQLString
    },
    cost: {
      type: GraphQLFloat
    },
    starring: {
```

Назначает объекту имя «movie», чтобы его можно было использовать в двух запросах

Определяет все поля с соответствующими типами

Использует формат с плавающей точкой для затрат на съемку

```
      type: new GraphQLList(new GraphQLObjectType({
        name: 'starring',
        fields: {
          name: {
            type: GraphQLString
          }
        }
      }))
    }
  }
})

module.exports = new GraphQLSchema({
  query: new GraphQLObjectType({
    name: 'Query',
    fields: {
      movies: {
        type: new GraphQLList(movie),
        resolve: () => movies
      },
      movie: {
        type: movie,
        args: {
          index: {
            type: GraphQLInt
          }
        },
        resolve: (r, {index}) => movies[index - 1]
      }
    }
  })
})
```

Возвращает весь массив фильмов

Возвращает только один фильм с использованием индекса (из параметра URL)

А теперь вернемся к клиентской части и посмотрим, как обратиться с запросом к этому аккуратному маленькому серверу.

15.2.4. Запросы к API и сохранение ответа в хранилище

Чтобы получить список фильмов, следует обратиться с запросом к серверу. После получения ответа его необходимо передать в хранилище. Эта операция выполняется асинхронно, и в ней задействован запрос HTTP, поэтому пришло время взяться за `axios`.

ОБЕЩАНИЯ И ОБРАТНЫЕ ВЫЗОВЫ

Библиотека axios реализует запросы HTTP с использованием обещаний (promises). Это означает, что сразу же после вызова функции возвращается обещание. Так как немедленное выполнение запроса HTTP не гарантировано, придется ожидать, пока обещание будет *выполнено*.

Для получения данных из обещания после его выполнения используется свойство `then`. Оно принимает функцию обратного вызова, которая получает один аргумент, и этот аргумент является результатом исходной операции — в данном случае вызова HTTP:

```
getPromise(options)
  .then((data)=>{
    console.log(data)
})
```

Использование обещания и обратного вызова (в then) — альтернатива для простого использования обратного вызова в том смысле, что приведенный код можно переписать без обещаний:

```
getResource(options, (data)=>{
  console.log(data)
})
```

По поводу обещаний нет единого мнения. Хотя некоторые разработчики предпочитают обещания и обратные вызовы из-за удобства синтаксиса обещаний `catch.all`, другие полагают, что обещания не стоят хлопот (я принадлежу к этому лагерю), особенно если учесть, что обещания могут «похоронить» ошибки и привести к сбою без выдачи информации. Тем не менее обещания являются частью ES6/ES2015 и никуда не исчезнут. Одновременно появляются новые паттерны, такие как генераторы и `async/await`, как следующий шаг эволюции в написании асинхронного кода[1].

Бесспорно, любое асинхронное программирование может ограничиваться простыми обратными вызовами. Но в большинстве современного кода (особенно кода клиентской части!) используются (или будут использоваться) обещания или async/await. По этой причине в книге используются обещания с `fetch()` и `axios`.

За дополнительной информацией об API обещаний обращайтесь к документации MDN (http://mng.bz/7DcO) и моей статье «Top 10 ES6 Features Every Busy JavaScript Developer Must Know» (https://webapplog.com/es6).

`axios`, как и `fetch()`, использует запросы на базе обещаний. Чтобы выполнить запрос HTTP GET, используйте свойство `get` объекта `axios`:

```
axios.get('/q')
```

Так как `axios` возвращает обещание, вы можете немедленно обратиться к его свойству `then`:

```
axios.get('/q').then(response => response)
```

[1] См. мои курсы по ES6 и ES7+ES8 по адресам https://node.university/p/es6 и https://node.university/p/es7-es8.

Теперь построим правильный запрос для GraphQL API. Для этого нужно использовать многострочный шаблон (обратите внимание: он использует обратные апострофы вместо одинарных кавычек):

```
axios.get(`/q?query={
  movie(index:1) {
    title,
    cover
  }
}`).then(response => this.props.fetchMovie(response))
```

Использование многострочного литерала (обратные кавычки) сохраняет внутренние разрывы строк, так что строка запроса будет состоять из нескольких логических строк. Нехорошо — новые строки в строке запроса могут нарушить URL-адреса конечной точки API. По этой причине необходимо удалить лишние пробелы и разрывы строк в вызовах HTTP, но оставить их в исходном коде. Библиотека `clean-tagged-string` (https://github.com/taxigy/clean-tagged-string) делает только это: она преобразует большой многострочный шаблон в меньший шаблон, не содержащий внутренних разрывов строк. Таким образом, следующий фрагмент:

```
clean`/q?query={
  movie(index:1) {
    title,
    cover
  }
}`
```

принимает следующий вид:

```
'/q?query={ movie(index:1) { title, cover } }'
```

Обратите внимание на синтаксис: за именем `clean` нет круглых скобок и оно смыкается со строкой шаблона. Это допустимый синтаксис *шаблонных литералов с тегами* (http://mng.bz/9CqH).

Теперь возьмем первый фильм с индексом 1:

```
const clean = require('clean-tagged-string').default

axios.get(clean`/q?query={
  movie(index:1) {
    title,
    cover
  }
}`).then(response => this.props.fetchMovie(response))
```

Теперь необходимо реализовать код получения любого фильма по его идентификатору. Также нужно запросить другие поля (не только `title` и `cover`), чтобы вывести представление на рис. 15.1. Полезно знать, что страница отдельного фильма не будет потеряна при перезагрузке, потому что вы добавили в `sendFile()` специальный серверный код с * для перехвата всех маршрутов, отправляющих index.html.

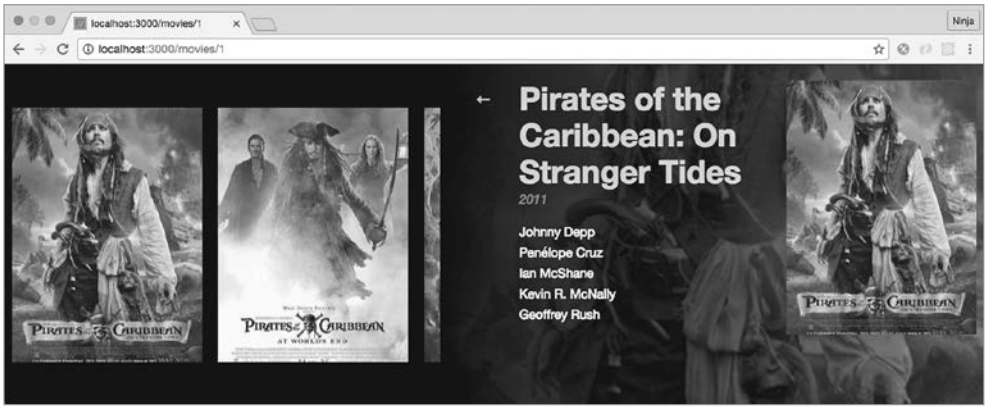


Рис. 15.1. Представление отдельного фильма от сервера Express (порт 3000) с поддержкой истории браузера (без символов #!)

Данные отдельного фильма можно получить от API в событии жизненного цикла, при этом используется наш любимый агент HTTP на базе обещаний — `axios`:

```
componentWillMount() {
  const query = clean`{
    movie(index:${id}) {
      title,
      cover,
      year,
      starring {
        name
      }
    }
  }`

  axios.get(`/q?query=${query}`)
    .then(response =>
      this.props.fetchMovie(response)
    )
}
```

Список запрошенных свойств для фильмов немного длиннее: это не только `title` и `cover`, но и `year` и `starring`. Так как `starring` представляет собой массив объектов, также необходимо объявить, какие свойства каких объектов вы хотите запросить. В данном случае нужно только `name`.

Ответ от API переходит к создателю действия `fetchMovie`. После этого хранилище обновляется тем фильмом, который был запрошен пользователем.

Выполняем связывание:

```
const {
  fetchMovieActionCreator
```

```
} = require('modules/movies.js')
...
module.exports = connect(({movies}) => ({
  movie: movies.current
}), {
  fetchMovie: fetchMovieActionCreator
})(Movie)
```

И рендер:

```
render() {
  const {
    movie = {
      starring: []
    }
  } = this.props;
  return (
    <div>
      <img src={`url(${movie.cover})`} alt={movie.title} />
      <div>
        <div>{movie.title}</div>
        <div>{movie.year}</div>
        {movie.starring.map((actor = {}, index) => (
          <div key={index}>
            {actor.name}
          </div>
        ))}
      </div>
      <Link to="/movies">
        ?
      </Link>
    </div>
  )
}
```

Чтобы улучшить структуру кода, добавим в компонент `Movie` метод `fetchMovie()`, уже знакомый по главе 14 (ch14/redux-netflix/src/components/movie/movie.js). Этот новый метод будет использоваться для AJAX-вызовов, которые, в свою очередь, передают действия. Метод находится в компоненте `Movie` (ch15/redux-graphql-netflix/client/components/movie/movie.js).

Листинг 15.4. Метод класса компонента fetchMovie()

```
// ...
fetchMovie(id = this.props.params.id) {
  const query = clean`{
    movie(index:${id}) {
      title,
      cover,
      year,
      starring {
        name
      }
```

Использует параметр React Router из URL для назначения идентификатора

Формирует запрос с использованием идентификатора, строки шаблона и clean

```
    }
  }`

  axios.get(`/q?query=${query}`)
    .then(response => {  ◄── Выдает запрос к /q
      this.props.fetchMovie(response)  ◄── Передает действие с данными от сервера
    }
  )
}
// ...
```

А теперь перейдем к получению списка фильмов.

15.2.5. Вывод списка фильмов

При выводе списка фильмов запрос к API выглядит по-другому, и отображение данных производится несколько иначе, нежели для одного фильма. Данные получаются от сервера GraphQL действительным запросом GraphQL с использованием асинхронного запроса GET, выполняемого с библиотекой `axios`, и помещаются в хранилище посредством действия. Следующее, что нужно сделать, — вывести эти данные для пользователя.

Вы уже знаете, что для получения данных от хранилища компонент должен быть связан с хранилищем, то есть упакованным в вызов функции `connect()`, отображающей состояние и действия в свойствах. В функции `render()` компонента используются свойства компонента. Но этим свойствам необходимы значения; вот почему используются вызовы AJAX/XHR, обычно после того, как компонент был подключен в первый раз (события жизненного цикла!).

Давайте объявим компонент, который будет получать данные из хранилища, извлекать их из свойств и выводить для пользователя. Сначала свяжем компонент с хранилищем (фрагмент взят из файла ch15/redux-graphql-netflix/client/components/movies/movies.js):

```
const React = require('react')
const { connect } = require('react-redux')
const {
  fetchMoviesActionCreator
} = require('modules/movies')

class Movies extends Component {
  // ...
}

module.exports = connect(({movies}) => ({
  movies: movies.all
}), {
  fetchMovies: fetchMoviesActionCreator
})(Movies)
```

Функция `connect()` получает два аргумента: первый связывает хранилище со свойствами компонента, а второй отображает создатели действий на свойства компонента. После этого в компоненте появляются два новых свойства: `this.props.movies` и `this.props.fetchMovies()`.

Теперь загрузим эти фильмы и при получении данных поместим их в хранилище с использованием создателя действия (передачи действия). Обычно данные могут запрашиваться через удаленный API в начале жизненного цикла компонента (`componentWillMount()` или `componentDidMount()`):

```
const {
  fetchMoviesActionCreator
} = require('modules/movies.js')
...
class Movies extends React.Component {
  componentWillMount() {
    const query = clean`{
      movies {
        title,
        cover
      }
    }`

    axios.get(`/q?query=${query}`)
      .then(response => {
        this.props.fetchMovies(response)
      }
    )
  }
// ...
}
module.exports = connect(({movies}) => ({
  movies: movies.all
}), {
  fetchMovies: fetchMoviesActionCreator
})(Movies)
```

Импортирует создатель действия

Передает действие для обновления хранилища ответом от сервера

Позволяет передать действие, предоставленное создателем действия

Наконец, выполняется рендер компонента `Movies` с использованием данных из свойств, полученных из хранилища Redux:

```
// ...
render() {
  const {
    movies = []
  } = this.props

  return (
    <div>
      {movies.map((movie, index) => (
        <Link
          key={index}
```

```
          to={`/movies/${index + 1}`}>
          <img src={`url(${movie.cover})`} alt={movie.title} />
        </Link>
      ))}
    </div>
  )
}
// ...
```

У каждого фильма есть свойства `cover` и `title`. Ссылка на фильм фактически обозначает его позицию в массиве фильмов. Для коллекций с тысячами элементов эта конфигурация не идеальна, потому что порядок в общем случае не гарантирован, но для наших целей этого достаточно. (Правильнее было бы использовать уникальный идентификатор, который автоматически генерируется такими базами данных, как MongoDB.)

Теперь компонент выводит список фильмов, хотя стилевое оформление в нем пока отсутствует. Обратитесь к исходному коду этой главы, чтобы увидеть, как он работает со стилями и трехуровневой иерархией компонентов.

15.2.6. GraphQL: заключение

На простейшем уровне поддержка GraphQL добавляется вполне элементарно и прозрачно. GraphQL несколько отличается от типичных REST-совместимых API: вы можете запросить любое свойство на любом уровне вложенности для любого подмножества сущностей, предоставляемых API. Тем самым обеспечивается эффективная работа с базами данных сложных объектов в GraphQL, тогда как в REST-архитектурах для получения тех же данных обычно требуется несколько запросов.

GraphQL — перспективная схема реализации взаимодействий «сервер — клиент». Она предоставляет больше возможностей для управления клиенту, который может диктовать структуру данных REST API. Инверсия управления позволяет разработчикам клиентской части запрашивать только те данные, которые им нужны, и при этом им не приходится изменять код серверной части (или обращаться с такой просьбой к команде разработки серверной части).

15.3. Вопросы

1. Какая из перечисленных команд используется для создания схемы GraphQL: `new graphql.GraphQLSchema()`, `graphql.GraphQLSchema()` или `graphql.getGraphQLSchema()`?
2. Вызовы API можно размещать в редьюсерах. Да или нет? (Подсказка: см. врезку **СОВЕТ** в главе 14.)

3. В какой из перечисленных методов включается вызов GraphQL для загрузки фильма: `componentDidUnmount()`, `componentWillMount()` или `componentDidMount()`?
4. Для GraphQL использовался следующий URL: `` `/q?query=${query}` ``. Что обозначает этот синтаксис — встроенную разметку, комментарий, шаблонный литерал, строковый шаблон, строковую интерполяцию?
5. `GraphQLString` — специальный тип схемы GraphQL, и этот класс находится в пакете `graphql`. Да или нет?

15.4. Итоги

- GraphQL — мощный и надежный механизм передачи данных клиентской части. Также он избавляет от необходимости писать большой объем кода серверной части.
- Чтобы включить поддержку истории браузера и URL без идентификаторов фрагмента в React Router, можно использовать вызов `sendFile()` в маршруте Express * для отправки index.html.
- Чтобы использовать Express не только как провайдера данных/API, но и как статический веб-сервер, используйте `express.static` с `app.use()`.
- URL в GraphQL имеют структуру `/q?query=...`, где *query* — ваш запрос данных.

15.5. Ответы

1. `new graphql.GraphQLSchema()`.
2. Нет. Размещать вызовы API в редьюсерах не рекомендуется. Лучше размещать их в компонентах (контейнерных компонентах, если точнее).
3. `componentWillMount()`, но `componentDidMount()` тоже подойдет. `componentDidUnmount()` не является допустимым методом жизненного цикла.
4. Шаблонный литерал, строковый шаблон и строковая интерполяция — все эти термины могут применяться для определения запроса в переменной.
5. Да. Это действительно код: `const {GraphQLString} = require('graphql');`. См. листинг 15.3.

16 Модульное тестирование кода React с Jest

Посмотрите вступительный видеоролик к этой главе, отсканировав QR-код или перейдя по ссылке http://reactquickly.co/videos/ch16.

ЭТА ГЛАВА ОХВАТЫВАЕТ СЛЕДУЮЩИЕ ТЕМЫ:

- Почему именно Jest?
- Модульное тестирование с Jest.
- Тестирование UI с Jest и TestUtils.

В современной разработке ПО тестирование играет важнейшую роль. Оно не менее важно, чем использование методов гибкой (Agile) разработки, написание хорошо документированного кода и достаточный запас кофе, — иногда даже и важнее. Качественное тестирование избавит вас от многих часов отладки в будущем. Код не актив, а пассив, поэтому вы должны по возможности упростить его сопровождение.

КОД — ПАССИВ?

Поиск в Google оригинала фразы «код не актив, а пассив»[1] дает около 191 миллиона результатов, так что определить ее происхождение будет сложно. Я не могу найти автора, но могу объяснить суть: в процессе разработки ПО вы строите приложения/продукты/сервисы, которые являются активами, но ваш код в их число не входит.

Активы приносят доход. Код сам по себе дохода не приносит. Да, код делает возможной работу продуктов, но это всего лишь инструмент для создания продуктов (которые являются активами). Сам по себе код активом не является — скорее

[1] Code isn't an asset, it's liability.

это необходимое зло для достижения конечной цели, то есть работоспособного приложения.

Таким образом, код является пассивом, потому что его нужно сопровождать. Большой объем кода не всегда автоматически означает больший доход или повышенное качество продукта; с другой стороны, большой объем кода почти всегда приводит к возрастанию сложности и затрат на сопровождение. Лучшие способы сокращения затрат на сопровождение — построение кода простого, надежного и достаточно гибкого для будущих изменений и усовершенствований. А тестирование, особенно автоматизированное, помогает при внесении изменений, потому что вы можете быть уверены в том, что изменения не нарушат работоспособность вашего приложения.

Применение тестирования при разработке (TDD[1]/BDD[2], Test-Driven/Behavior-Driven Development) может упростить сопровождение. Кроме того, оно повысит конкурентоспособность вашего приложения — итерации будут проходить быстрее, а уверенность в том, что ваш код работает, сделает вашу работу более продуктивной.

ПРИМЕЧАНИЕ Исходный код примеров этой главы доступен по адресам www.manning.com/books/react-quickly и https://github.com/azat-co/react-quickly/tree/master/ch16. Также некоторые демонстрационные примеры доступны по адресу http://reactquickly.co/demos.

16.1. Разновидности тестирования

Существуют несколько разновидностей тестирования. Чаще всего они делятся на три категории: модульное, интеграционное и UI-тестирование (рис. 16.1). Ниже приведена краткая сводка всех категорий от нижнего уровня к верхнему:

- *Модульное тестирование* — система тестирует автономные методы и классы. Зависимости или взаимосвязанные части либо отсутствуют, либо их количество минимально. Чтобы убедиться в том, что метод работает как нужно, должно быть достаточно тестируемого кода. Например, чтобы протестировать модуль, генерирующий случайные пароли, можно вызвать метод из модуля и сравнить результат с шаблоном регулярного выражения. К этой категории также относятся тесты, в которых один блок функциональности строится совместными усилиями нескольких частей или модулей. Например, несколько компонентов могут совместно предоставлять функциональность ввода пароля с проверкой его надежности. Чтобы протестировать эту функциональность, можно подать значение на один компонент (вход) и отслеживать изменения в проверке надежности (достаточно или нет). Согласно отраслевой практике, эта категория должна включать около 70 % ваших тестов (рис. 16.1), и она определенно превосходит по численности все остальные разновидности тестирования.

[1] Https://ru.wikipedia.org/wiki/Разработка_через_тестирование.
[2] Https://ru.wikipedia.org/wiki/BDD_(программирование).

- *Интеграционное тестирование* — тесты обычно включают другие зависимости и требуют специальной тестовой среды. Интеграционные тесты должны составлять около 20 % всех тестов. После того как у вас сформируется прочная основа из модульных тестов и подтверждение в виде функциональных тестов, лишние интеграционные тесты нежелательны, потому что их сопровождение только замедлит разработку. При каждом изменении пользовательского интерфейса интеграционные тесты придется обновлять. Часто это приводит к ненадежности UI-тестов и полному отказу от интеграционного тестирования, что еще хуже.
- *UI-тестирование* — тесты часто воспроизводят пользовательские истории гибкой разработки и/или подразумевают тестирование всей системы, в котором очевидным образом задействованы все мыслимые зависимости и сложности. UI-тесты менее устойчивы и более сложны (то есть затратны) в сопровождении, поэтому они должны составлять всего лишь около 10 % от общего количества тестов.

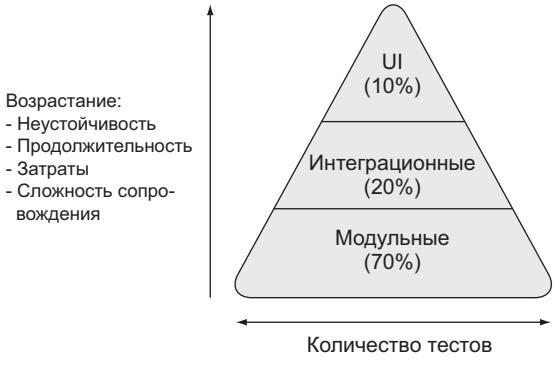


Рис. 16.1. Пирамида тестирования согласно рекомендованным практикам разработки ПО

В этой главе рассматривается модульное тестирование приложений React, а также отчасти UI-тестирование компонентов React с использованием фиктивного рендера DOM в React и Jest. Также будет использоваться стандартный инструментарий Node, npm, Babel и Webpack. Знакомство с темой модульного тестирования начнется с Jest.

16.2. Почему именно Jest?

Jest (https://facebook.github.io/jest) — инструмент командной строки, построенный на основе Jasmine. Его интерфейс похож на интерфейс Jasmine. Если вы уже работали с Mocha, вы увидите, что Jest выглядит похоже и легко осваивается. Разработкой Jest занимается Facebook, и Jest часто используется вместе с React; документация API доступна по адресу https://facebook.github.io/jest/docs/api.html#content.

Основные особенности Jest:

- Мощный механизм имитации (https://facebook.github.io/jest/docs/mock-functions.html) модулей JavaScript/Node упрощает изоляцию кода для модульного тестирования.
- Начало работы требует меньшей подготовки по сравнению с другими системами выполнения тестов, например: Mocha требует импортирования Chai или автономного Expect. Jest также автоматически находит тесты в папке __tests__.
- Тесты могут выполняться изолированно (sandboxed) и параллельно, что ускоряет их выполнение[1].
- Вы можете провести статический анализ кода при поддержке Facebook Flow (https://flowtype.org) — системы статической проверки типов для JS.
- Jest обеспечивает модульность, разнообразную настройку и адаптируемость (через поддержку тестовых утверждений Jasmine).

ИМИТАЦИЯ, СТАТИЧЕСКИЙ АНАЛИЗ И JASMINE

Термин «имитация» (mocking) означает моделирование некоторой части зависимости для тестирования текущего кода. В Jest до версии 15[2] все импортируемые зависимости имитировались автоматически, что может быть полезно при частом применении имитации. Многим разработчикам автоматическая имитация не нужна, поэтому в Jest версий 15+ она отключена по умолчанию, хотя ее можно включить при необходимости.

Термин «статический анализ» означает, что код может быть проанализирован перед выполнением, а это обычно включает проверку типов. Flow — библиотека, добавляющая проверку типов в нетипизованный (более или менее) JavaScript.

Jasmine — полнофункциональный фреймворк тестирования с языком тестовых утверждений (assertions). Jest во внутренней реализации расширяет Jasmine, чтобы вам не приходилось ничего импортировать и настраивать. Таким образом, вы получаете лучшее с обеих сторон: популярный интерфейс Jasmine без лишних зависимостей или настройки.

Существуют разные мнения относительно того, какой тестовый фреймворк лучше подходит для той или иной работы. Многие проекты используют фреймворк Mocha, обладающий широкой функциональностью. Фреймворк Jasmine вырос из разработки клиентской части, но обладает взаимозаменяемостью с Mocha и Jest. Все они используют одни и те же конструкции для определения тестов и их наборов[3]:

---

[1] Christopher Pojer, «JavaScript Unit Testing Performance», Jest, 11 марта 2016 г., http://mng.bz/YfXz.

[2] См. Christoph Pojer, «Jest 15.0: New Defaults for Jest», 1 сентября 2016 г., http://mng.bz/p20n.

[3] См. Christoph Pojer, «Jest 15.0: New Defaults for Jest», 1 сентября 2016 г., http://mng.bz/p20n.

- `describe` — набор тестов.
- `it` — тестовый случай.
- `before` — подготовка.
- `beforeEach` — подготовка для каждого набора или тестового случая.
- `after` — завершающие действия.
- `afterEach` — завершающие действия для каждого набора или случая.

Не углубляясь в жаркие споры о том, какой фреймворк лучше остальных, я предлагаю относиться непредвзято и присмотреться к Jest из-за перечисленных возможностей и из-за того, что этот фреймворк разработан в том же сообществе, которое разрабатывает React. Таким образом вы сможете принять более обоснованное решение относительно того, какой фреймворк стоит использовать в вашем следующем проекте React.

Многие современные фреймворки, такие как Mocha, Jasmine и Jest, похожи для многих задач. Различия зависят от ваших личных предпочтений (возможно, вам нравится автоматическая имитация, а может, не нравится) и от особых требований конкретного проекта (нужна ли вам вся функциональность, предоставляемая Mocha, или хватит чего-то более легковесного: например, Test Anything Protocol [TAP, https://testanything.org] или node-tap [www.node-tap.org]?). Jest — хорошая отправная точка, потому что, научившись использовать Jest со служебными средствами и методами React, вы сможете использовать другие системы запуска тестов и тестовые фреймворки, такие как Mocha, Jasmine и node-tap.

16.3. Модульное тестирование с Jest

Если вы никогда не работали с тестовыми фреймворками, которые я упоминал, не беспокойтесь: Jest изучить несложно. Основная команда `describe` определяет набор тестов, который используется в качестве контейнера для тестов, а команда `it` определяет отдельный тест, называемый *тестовым случаем*. Тестовые случаи заключены в набор тестов.

Другие конструкции, такие как `before`, `after` и их `Each`-собратья `beforeEach` и `afterEach`, выполняются либо после, либо до набора тестов или тестового случая. При добавлении `Each` фрагмент кода выполняется многократно вместо одного раза.

Написание тестов состоит из создания наборов тестов, тестовых случаев и тестовых утверждений. Тестовые утверждения напоминают вопросы «да/нет» в удобном формате (BDD).

Пример (пока без тестовых утверждений):

```
describe('Noun: method or a class/module name', () => {
  before((done) => {  ◄——  Определяет обратный вызов done ()
```

```
    // This code will be called just once before all it statements
    done()  ◄——  Вызывается done (), когда тестирование асинхронного кода завершено
  })
  beforeEach((done) => {
    // This code will be called many times before all it statements
    done()
  })
  it('Verb describing the behavior', (done) => {
    // Тестовые утверждения
    done()
  })
  it('Verb describing the behavior', (done) => {
    // Тестовые утверждения
    done()
  })
  ...
  after((done) => {
    // Этот код будет выполнен только один раз после всех команд it
    done()
  })
  afterEach((done) => {
    // Этот код будет выполнен много раз после всех команд it
    done()
  })
})
```

Пример должен содержать как минимум по одной команде `describe` и `it`, но их максимальное количество не ограничено. Все остальные команды, такие как `before` и `after`, не являются обязательными.

Никакие компоненты React еще не тестируются. Прежде чем работать с компонентами React, необходимо больше узнать о Jest. Для этого мы поработаем над примером Jest, не имеющим пользовательского интерфейса.

В этом разделе мы создадим и протестируем модуль, генерирующий случайные пароли. Представьте, что вы работаете над страницей регистрации нового пользователя в приложении чата. Для этого понадобится функция генерирования паролей, верно? Модуль будет автоматически генерировать случайные пароли. Чтобы не усложнять задачу, пароли будут состоять из восьми алфавитно-цифровых символов. Проект (модуль) имеет следующую структуру:

```
/generate-password
  /__test__
    generate-password.test.js
  /node_modules
    generate-password.js
    package.json
```

Мы используем синтаксис модулей CommonJS/Node, который широко поддерживается в Node, а также в разработке браузерного кода с использованием Browserify и Webpack. Код модуля хранится в файле ch16/generate-password.js.

Листинг 16.1. Модуль генерирования паролей

```
module.exports = () => {
  return Math.random().toString(36).slice(-8)
}
```

Использует сегмент с отрицательным числом для изменения направления (справа налево)

Напомню, что в этом файле функция экспортируется глобальной конструкцией `module.exports`. Это синтаксис Node.js и CommonJS. Вы можете использовать его в браузере с такими дополнительными инструментами, как Webpack и Browserify (http://browserify.org).

Функция генерирует число при помощи `Math.random()` и преобразует его в строку. Длина строки равна 8 символам, как указано в вызове `slice(-8)`.

Чтобы протестировать модуль, выполните следующую команду Node в терминале. Она импортирует модуль, вызывает его функцию и выводит результат:

```
node -e \"console.log(require('./generate-password.js')())\"
```

Этот модуль можно усовершенствовать, заставив его работать с другим количеством символов (не только 8).

16.3.1. Написание модульных тестов в Jest

Чтобы начать использовать Jest, создайте новую папку проекта и выполните команду `npm init`, чтобы создать файл package.json. Если система npm недоступна, сейчас самое время установить ее; выполните инструкции в приложении Б.

После того как файл package.json будет создан в новой папке, установите Jest:

```
$ npm install jest-cli@19.0.2 --save-dev --save-exact
```

Я использую `jest-cli` версии 19.0.2; убедитесь в том, что вы используете ту же или совместимую версию. Ключ `--save-dev` добавляет запись в файл package.json. Откройте файл и вручную замените запись `test` на `jest`, как показано в листинге 16.2 (ch16/jest/package.json). Таким образом будет добавлена команда тестирования. Также добавьте сценарий `start`:

Листинг 16.2. Сохранение тестовой команды CLI

```
{
  "name": "jest",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "jest",
    "start": "node -e
```

Заменяет тестовый сценарий по умолчанию на jest

```
    ➥ \"console.log(require('./generate-password.js')())\""
  },
  "author": "Azat Mardan",
  "license": "MIT",
  "devDependencies": {
    "jest-cli": "19.0.2"
  }
}
```

Сохраняет команду Node для получения случайного пароля

Использует версию 19.0.2 без ^, чтобы использовалась в точности указанная версия

Создайте папку с именем __tests__. Имя важно, потому что Jest берет тесты из этой папки. Создайте свой первый тест Jest в файле __tests__/generate-password.js.

Как правило, имитируются только те зависимости, которые не нужны для изоляции тестируемой библиотеки. Jest до версии 15 автоматически имитирует каждый необходимый файл, так что вам придется использовать `dontMock()` или `jest.autoMockOff()`, чтобы отключить это поведение для главного тестируемого файла (generate-password.js). Это делается примерно так:

```
jest.dontMock('../generate-password.js')
```

К счастью, для версии Jest, используемой в этой главе (v19), отключать автоматическую имитацию не нужно, потому что она отключена по умолчанию. А значит, вы можете пропустить строку кода `dontMock()` или оставить ее закомментированной.

Тестовый файл содержит один набор тестов (единственный описанный командой `describe`), который проверяет, что значение соответствует шаблону регулярного выражения `/^[a-z0-9]{8}$/` (только алфавитно-цифровые символы и ровно 8 символов) для проверки надежного пароля (ch16/generate-password/__tests__/generate-password.test.js). Пароли пользователей вашего чата должны быть защищены от простого перебора!

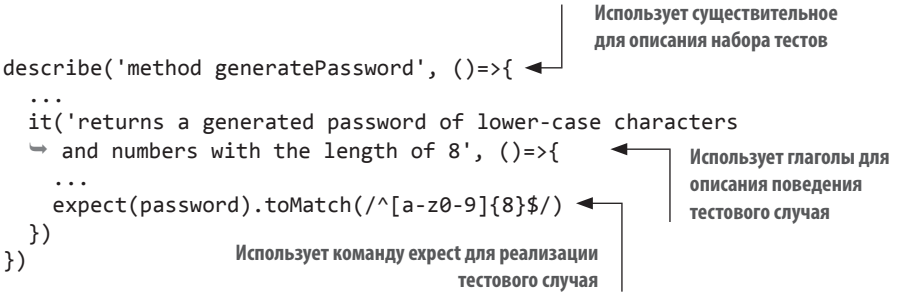
Листинг 16.3. Тестовый файл для модуля генерирования пароля

```
describe('method generatePassword', ()=>{
  let password
  generatePassword = require('../generate-password')
  it('returns a generated password of lower-case characters
  ➥ and numbers with the length of 8', (done)=>{
    password = generatePassword()
    expect(password).toMatch(/^[a-z0-9]{8}$/)
    done()
  })
})
```

Использует require — специальную глобальную конструкцию Node.js, которая импортирует модуль в файл script.j

Вызывает done(), если вы определили аргумент, необходимый для асинхронных тестов и необязательный для синхронных

Тест можно запустить командой `$ npm test`. На терминал выводится результат, который выглядит примерно так:



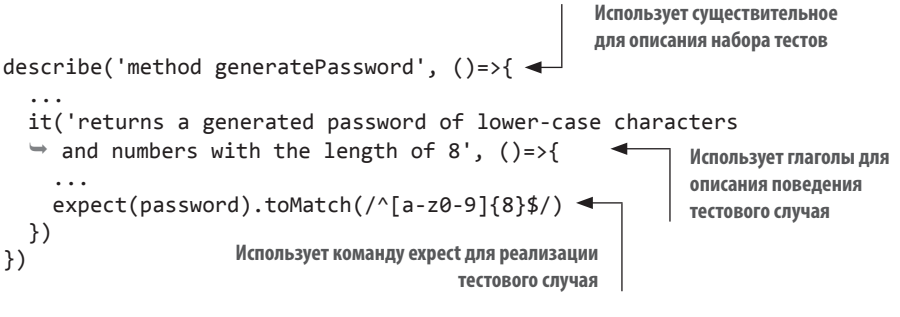
16.3.2. Тестовые утверждения Jest

По умолчанию Jest использует синтаксис BDD, в основе которого лежит синтаксис Expect (https://facebook.github.io/jest/docs/api.html). Expect — популярный язык, заменяющий тестовые утверждения TDD. Он существует в нескольких вариантах, и Jest использует несколько упрощенную версию (на мой взгляд). В отличие от других фреймворков (например, Mocha), где для поддержки синтаксиса приходится устанавливать дополнительные модули, в Jest это происходит автоматически.

TDD И BDD

Термин «TDD» может обозначать разработку через тестирование (Test-Driven Development) или синтаксис TDD с тестовыми утверждениями. Вкратце: в ходе разработки через тестирование вы пишете тест, запускаете его (он не проходит), обеспечиваете его прохождение, а затем совершенствуете его структуру (рефакторинг).

Безусловно, разработка через тестирование может выполняться и с BDD. Главное преимущество стиля BDD заключается в том, что он ориентирован на взаимодействие с каждым участником межфункциональной команды, не только с разработчиком. TDD скорее напоминает технический жаргон. Формат BDD упрощает чтение тестов — в идеале из заголовка должно быть видно, что вы тестируете, как в следующем примере:



Ниже перечислены основные методы Expect, поддерживаемые в Jest (также есть много других). Вы передаете `expect()` фактические значения (возвращаемые программой) и используете следующие методы для сравнения этих значений с ожидаемыми, жестко зафиксированными в тестах:

- `.not` — инвертирует следующее сравнение в цепочке.
- `expect(OBJECT).toBe(value)` — проверяет на равенство value с использованием тройного знака равенства JavaScript `===` (то есть проверяется значение и тип, а не только значение[1]).
- `expect(OBJECT).toEqual(value)` — проверяет value на глубокое равенство[2].
- `expect(OBJECT).toBeFalsy()` — проверяет значение на квазиложность (см. врезку).
- `expect(OBJECT).toBeTruthy()` — проверяет значение на квазиистинность.
- `expect(OBJECT.toBeNull()` — проверяет значение на равенство `null`.
- `expect(OBJECT).toBeUndefined()` — проверяет значение на неопределенность.
- `expect(OBJECT).toBeDefined()` — проверяет значение на определенность.
- `expect(OBJECT).toMatch(regexp)` — проверяет значение на соответствие регулярному выражению.

КВАЗИИСТИННОСТЬ И КВАЗИЛОЖНОСТЬ

В JavaScript/Node квазиистинное (`truthy`) значение интерпретируется как `true` в логическом контексте в команде `if/else`. С другой стороны, квазиложное (Falsy) значение интерпретируется как `false` в `if/else`.

Согласно официальному определению, квазиистинным является любое значение, которое не является квазиложным. При этом определено всего шесть ложных значений:

- `false`
- `0`
- `""` (пустая строка)
- `null`
- `undefined`
- `NaN` («не число»)

Любое значение, не входящее в этот список, является квазиистинным.

[1] См. «Equality Comparisons and Sameness», Mozilla Developer Network, http://mng.bz/kliO.

[2] При проверке глубокого равенства объекты, включая все их свойства и значения, сравниваются до максимального уровня вложенности (в глубину). Стандартного API в JavaScript для такой проверки не существует, но есть такие реализации, как модуль Node assert (http://mng.bz/rhoX) и deep-equal (www.npmjs.com/package/deep-equal).

Итак, Jest можно использовать для модульных тестов, которые должны составлять самую многочисленную категорию тестов. Работая на низком уровне, они обладают большей надежностью и устойчивостью, что снижает затраты на их сопровождение.

К настоящему моменту мы создали модуль и протестировали его метод с Jest. Это типичный модульный тест. Никакие зависимости в нем не задействованы — только сам тестируемый модуль. Этот урок должен был подготовить вас к тестированию компонентов React. А теперь рассмотрим более сложный процесс UI-тестирования. В следующем разделе рассматриваются средства тестирования React, позволяющие проводить UI-тестирование.

16.4. UI-тестирование React с использованием Jest и TestUtils

В общем случае UI-тесты (которые, как рекомендуется, составляют 10 % ваших тестов) тестируют ваши компоненты, их поведение и даже целые деревья DOM. Конечно, компоненты можно тестировать вручную, но это просто ужасная идея! Человеку свойственно ошибаться, а тестирование занимает слишком много времени. Ручное UI-тестирование должно быть сведено к минимуму.

Как насчет автоматизации UI-тестирования? Для этой цели можно воспользоваться консольным браузером (https://en.wikipedia.org/wiki/Headless_browser), который работает как настоящий браузер, но не имеет графического интерфейса. Так тестируется большинство приложений Angular 1. Этот процесс также можно использовать с React, но он непрост, часто работает медленно и требует значительных вычислительных мощностей.

Другой метод автоматизации UI-тестирования использует виртуальную модель DOM в React, для работы с которой используется тестовая среда JavaScript, сходная с браузером и реализованная jsdom (https://github.com/tmpvar/jsdom). Для использования виртуальной модели DOM в React вам понадобится инструмент, тесно связанный с библиотекой React Core, но не входящий в нее: TestUtils — служебная программа React для тестирования ее компонентов. Проще говоря, TestUtils позволяет создать компонент и выполнить его рендеринг в фиктивной модели DOM. Далее вы можете экспериментировать, обращаясь к элементам по тегам и классам. Все это делается из командной строки без необходимости использования браузера (консольного или нет).

> **ПРИМЕЧАНИЕ** Также существуют другие дополнения React, список которых доступен по адресу https://facebook.github.io/react/docs/addons.html. Многие из них более не разрабатываются или все еще находятся на экспериментальной стадии, что на практике означает, что команда React может изменить их интерфейс или прекратить поддержку. Имена всех этих дополнений строятся по схеме `react-addons-NAME`. TestUtils относится к числу дополнений и, как и другие дополнения React, устанавливается через npm. (Использовать TestUtils без npm невозможно; если эта программа у вас еще не установлена, обратитесь к инструкциям в приложении А.)

Для React версий ранее 15.5.4 программа TestUtils существовала в виде пакета npm с именем `react-addons-test-utils` (https://facebook.github.io/react/docs/test-utils.html). Например, если вы использовали React версии 15.2.1, для установки `react-addons-test-utils` версии 15.2.1 из npm использовалась следующая команда:

```
$ npm install react-addons-test-utils@15.2.1 --save-dev --save-exact
```

А в исходный код теста включается следующая строка (React до версии 15.5.4):

```
const TestUtils = require('react-addons-test-utils')
```

В React версии 15.5.4 ситуация немного упростилась, потому что TestUtils находится в ReactDOM (`react-dom` в npm). Устанавливать отдельный пакет для этого примера не нужно, потому что вы используете более новую версию:

```
const TestUtils = require('react-dom/test-utils')
```

TestUtils содержит несколько основных методов для рендера компонентов, моделирования таких событий, как `click`, `mouseOver` и т. д., и поиска элементов в отрендеренном компоненте. Начнем с рендеринга, а остальные методы будут представлены по мере надобности.

Для демонстрации метода `render()` в TestUtils листинг 16.4 рендерит элемент в переменной `div` без использования консольного (или обычного, если уж на то пошло) браузера (ch16/testutils/__tests__/render-props.js).

Листинг 16.4. Рендеринг элемента React в Jest

```
describe('HelloWorld', ()=>{
  const TestUtils = require('react-dom/test-utils')
  const React = require('react')

  it('has props', (done)=>{

    class HelloWorld extends React.Component {
      render() {
        return <div>{this.props.children}</div>
      }
    }
    let hello = TestUtils.renderIntoDocument(<HelloWorld>Hello Node!
    ➥ </HelloWorld>)
    expect(hello.props).toBeDefined()
    console.log('my hello props:', hello.props) // my div: Hello Node!

    done()
  })
})
```

А файл package.json для примера ch16/testutils с Babel, Jest CLI, React и React DOM выглядит примерно так:

```
{
  "name": "password",
  "version": "2.0.0",
  "description": "",
  "main": "index.html",
  "scripts": {
    "test": "jest",
    "test-watch": "jest --watch",
    "build-watch": "./node_modules/.bin/webpack -w",
    "build": "./node_modules/.bin/webpack"
  },
  "author": "Azat Mardan",
  "license": "MIT",
  "babel": {
    "presets": [
      "react"
    ]
  },
  "devDependencies": {
    "babel-jest": "19.0.0",
    "babel-preset-react": "6.24.1",
    "jest-cli": "19.0.2",
    "react": "15.5.4",
    "react-dom": "15.5.4"
  }
}
```

ПРЕДУПРЕЖДЕНИЕ `renderIntoDocument()` работает только с пользовательскими компонентами, а не со стандартными компонентами DOM: `<p>`, `<div>`, `<section>` и т. д. Таким образом, если вы получаете ошибку вида «`Error: Invariant Violation: findAllInRenderedTree(...): instance must be a composite component`», убедитесь в том, что рендерингу подвергается нестандартный (ваш собственный) класс компонента, а не стандартный класс. За дополнительной информацией обращайтесь по адресам http://mng.bz/8AOc и https://github.com/facebook/react/issues/4692 в ветках обсуждения на GitHub.

После создания переменной `hello`, значением которой является дерево компонентов React (включающее все дочерние компоненты), вы можете проверить ее содержимое одним из методов поиска элементов. Например, можно получить элемент `<div>` из элемента `<HelloWorld/>`, как показано в листинге 16.5 (ch16/testutils/__tests__/scry-div.js).

Листинг 16.5. Поиск дочернего элемента элемента React

```
describe('HelloWorld', ()=>{
  const TestUtils = require('react-dom/test-utils')
  const React = require('react')

  it('has a div', (done)=>{

    class HelloWorld extends React.Component {
```

```
      render() {
        return <div>{this.props.children}</div>
      }
    }
    let hello = TestUtils.renderIntoDocument(
      <HelloWorld>Hello Node!</HelloWorld>
    )
    expect(TestUtils.scryRenderedDOMComponentsWithTag(
      hello, 'div'
    ).length).toBe(1)
    console.log('found this many divs: ',
      TestUtils.scryRenderedDOMComponentsWithTag(hello, 'div').length)

    done()
  })
})
...
```

`scryRenderedDOMComponentsWithTag()` позволяет получить массив элементов по именам тегов (например, `div`). Существуют ли другие средства получения элементов? Да!

16.4.1. Поиск элементов с TestUtils

Кроме `scryRenderedDOMComponentsWithTag()` существуют и другие средства получения списка элементов (с префиксом `scry` и `Components` во множественном числе) или отдельного элемента (с префиксом `find` и `Component` в единственном числе). В обоих случаях используется класс элемента, а не класс компонента — это не одно и то же (`btn`, `main` и т. д.).

Кроме имен тегов элементы можно получать по типу (класс компонента) или по классам CSS. Например, `HelloWorld` — тип, а `div` — имя тега (мы использовали его для получения списка критериев).

Комбинируя `scry` и `find` с `Class`, `Type` и `Tag`, можно получить шесть разных методов в зависимости от ваших потребностей. Ниже перечислены эти методы и возвращаемые ими результаты.

- `scryRenderedDOMComponentsWithTag()` — много элементов; вам известно их имя тега.
- `findRenderedDOMComponentWithTag()` — один элемент; вам известно его уникальное имя тега (другими словами, никакой другой элемент в компоненте не имеет такого же имени тега).
- `scryRenderedDOMComponentsWithClass()` — много элементов; вам известно их имя класса.
- `findRenderedDOMComponentWithClass()` — один элемент; вам известно его уникальное имя класса.

- `scryRenderedComponentsWithType()` — много элементов; вам известен их тип.
- `findRenderedComponentWithType()` — один элемент; вам известен его тип.

Как видите, в том, что касается извлечения необходимых элементов из компонентов, никакого дефицита методов нет. Если вам нужен совет, я бы порекомендовал использовать классы или типы (классы компонентов), потому что они повышают надежность выбора элементов. Например, представьте, что вы используете имена тегов, потому что сейчас существует только один элемент `<div>`. Если позднее вы решите добавить в свой код элементы с тем же именем тега (то есть более одного `<div>`), тест придется переписывать. Если вы используете класс HTML для тестирования `<div>`, тест будет нормально работать и после того, как вы добавите новые элементы `<div>` в тестируемый компонент.

Единственный случай, в котором может быть оправданно использование имен тегов, — необходимость тестирования всех элементов с конкретным именем тега (`scryRenderedDOMComponentsWithTag()`) или ваш компонент настолько мал, что других элементов с тем же именем тега не существует (`findRenderedDOMComponentWithTag()`). Например, если у вас имеется компонент без состояния, который содержит якорный тег `<a>`, к нему добавлено несколько классов HTML, и дополнительных якорных тегов уже не будет.

16.4.2. UI-тестирование виджета для генерирования паролей

Представьте UI-виджет, который используется на странице регистрации нового пользователя для автоматического генерирования пароля определенной надежности. Как видно из рис. 16.2, он содержит поле ввода, кнопку Generate и список критериев.

В следующем разделе будет рассмотрен весь проект, а пока мы сосредоточимся на пакете TestUtils и его интерфейсе. После того как TestUtils и другие зависимости (например, Jest) будут установлены, вы можете создать тестовый файл Jest для UI-тестирования вашего виджета; назовем его password/__tests__/password.test.js, потому что тестируется компонент для генерирования пароля.

Тест имеет следующую структуру:

```
describe('Password', function() {
  it('changes after clicking the Generate button', (done)=>{
    // Импортирование
    // Рендеринг
    // Тестовые утверждения для контента и поведения
    done()
  })
})
```

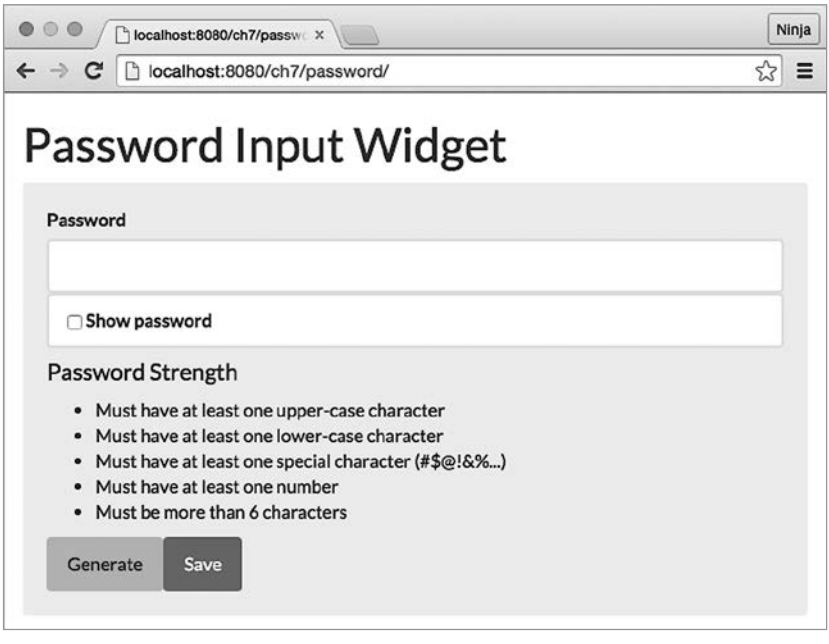


Рис. 16.2. Виджет, генерирующий пароль по заданным критериям надежности

Зависимости определяются в `describe`. Обратите внимание на создание сокращения `fD` для `ReactDOM.findDOMNode()`, потому что мы будем часто его использовать:

```
const TestUtils = require('react-dom/test-utils')
    const React = require('react')
    const ReactDOM = require('react-dom')
    const Password = require('../jsx/password.jsx')
    const fD = ReactDOM.findDOMNode
```

Для рендеринга компонента необходимо использовать `renderIntoDocument()`. В частности, так можно выполнить рендер компонента `Password` и сохранить ссылку на объект в переменной `password`. Передаваемые свойства будут ключами правил силы пароля. Например, `upperCase` требует наличия как минимум одного символа верхнего регистра:

```
let password = TestUtils.renderIntoDocument(<Password
      upperCase={true}
      lowerCase={true}
      special={true}
      number={true}
      over6={true}
      />
    )
```

Пример написан на JSX, потому что Jest автоматически использует `babel-jest` при установке этого модуля (`npm i babel-jest --save-dev`) и настраивает конфигурацию Babel на использование `"presets": ["react"]`. Вы не сможете использовать JSX в Jest, если не захотите включать `babel-jest`. В этом случае вызовите `createElement()`:

```
let password = TestUtils.renderIntoDocument(
    React.createElement(Password, {
      upperCase: true,
      lowerCase: true,
      special: true,
      number: true,
      over6: true
    })
  )
```

После того как будет выполнен рендеринг компонента вызовом `renderIntoDocument()`, вы сможете легко извлечь нужные элементы — дочерние элементы `Password` — и выполнить тестовые утверждения для проверки работоспособности вашего виджета. Вызовы получения элементов можно рассматривать как ваш персональный jQuery: вы можете использовать теги или классы. Как минимум тест должен включать следующие проверки:

1. Присутствует элемент `Password` со списком (`<li>`), пункты которого определяют критерии надежности.
2. Первый пункт в списке надежности содержит конкретный текст.
3. Второй пункт не выполняется (перечеркнутый текст).
4. Присутствует кнопка Generate (`generate-btn`) — щелкните на ней!
5. После щелчка на Generate второй пункт списка выполняется (становится видимым).

Кнопка Generate выполняет все критерии и делает пароль видимым (чтобы пользователи могли запомнить его), но код теста для этой возможности в книге не приводится. Он станет вашим домашним заданием на следующую неделю.

Начнем с (1). `TestUtils.scryRenderedDOMComponentsWithTag()` получает все элементы заданного класса. В данном случае это класс `li` для элементов `<li>`, потому что именно они будут использоваться списком критериев: `<ul><li>`. Вызов `toBe()`, работающий аналогично тройному знаку равенства (`===`), может использоваться для проверки того, что длина списка равна 5:

```
let rules = TestUtils.scryRenderedDOMComponentsWithTag(password, 'li')
expect(rules.length).toBe(5)
```

В случае второго пункта (первый пункт содержит конкретный текст) следует использовать `toEqual()`. Первый пункт должен сообщать, что пароль должен содержать символ верхнего регистра. Это один из критериев надежности пароля:

```
expect(fD(rules[0]).textContent).toEqual('Must have
➥ at least one uppercase character')
```

Чтобы проверить позиции (3), (4) и (5), следует найти кнопку, щелкнуть на ней и сравнить значения второго критерия (он должен переключиться с простого текста на перечеркнутый).

TOBE() И TOEQUAL()

`toBe()` и `toEqual()` в Jest работают по-разному. Проще всего запомнить, что `toBe()` эквивалентно `===` (строгое равенство), а `toEqual()` проверяет, что два объекта имеют одинаковое значение. Таким образом, оба тестовых утверждения будут выполняться:

```
const copy1 = {
  name: 'React Quickly',
  chapters: 19,
}
const copy2 = {
  name: 'React Quickly',
  chapters: 19,
}

describe('Two copies of my books', () => {
  it('have all the same properties', () => {
    expect(copy1).toEqual(copy2) // Выполняется
  })
  it('are not the same object', () => {
    expect(copy1).not.toBe(copy2) // Выполняется
  })
})
```

Но когда вы сравниваете литералы (скажем, число 5 и строку «`Must have at least one uppercase character`», `toBe()` и `toEqual()` возвращают один и тот же результат:

```
expect(rules.length).toBe(5) // Выполняется
expect(rules.length).toEqual(5) // Выполняется
expect(fD(rules[0]).textContent).toEqual('Must have
➥ at least one upper-case character') // Выполняется
expect(fD(rules[0]).textContent).toBe('Must have
➥ at least one upper-case character') // Выполняется
```

Существует метод `TestUtils.findRenderedDOMComponentWithClass()`, который похож на `TestUtils.scryRenderedDOMComponentsWithTag()`, но возвращает только один элемент; если элементов несколько, произойдет ошибка. А для имитации действий пользователя существует объект `TestUtils.Simulate`, методы которого напоминают имена событий в верблюжьем регистре: например, `Simulate.click`, `Simulate.keyDown` и `Simulate.change`.

Воспользуемся `findRenderedDOMComponentWithClass()` для получения кнопки, а затем воспользуемся `Simulate.click` для имитации щелчка. Все это делается в коде без браузера:

```
let generateButton =
➥ TestUtils.findRenderedDOMComponentWithClass(password, 'generate-btn')
    expect(fD(rules[1]).firstChild.nodeName.toLowerCase()).toBe('#text')
    TestUtils.Simulate.click(fD(generateButton))
    expect(fD(rules[1]).firstChild.nodeName.toLowerCase()).toBe('strike')
```

Этот тест проверяет, что компонент `<li>` содержит элемент `<strike>` (для вывода перечеркнутого текста) при щелчке на кнопке. Кнопка генерирует случайный пароль, который удовлетворяет второму критерию (`rules[1]`), как и всем остальным, а также содержит как минимум один символ нижнего регистра. Здесь работа закончена, можно переходить к другим тестам.

Вы уже видели объект `TestUtils.Simulate` в действии. Он может инициировать не только щелчки, но и другие взаимодействия — например, изменение значения в поле ввода или нажатие клавиши Enter (`keyCode` 13):

```
ReactTestUtils.Simulate.change(node)
ReactTestUtils.Simulate.keyDown(node, {
  key: "Enter",
  keyCode: 13,
  which: 13})
```

> **ПРИМЕЧАНИЕ** Данные, которые будут использоваться в компоненте (например, `key` или `keyCode`), приходится передавать вручную, потому что TestUtils не генерирует их автоматически. Для каждого действия пользователя, поддерживаемого React, в TestUtils существуют методы.

Ниже приведет файл манифеста проекта package.json. Он также включает библиотеку `shallow-rendering`, которая будет рассмотрена ниже. Чтобы выполнить примеры из `ch16/password`, установите зависимости `npm i`, а затем выполните команду `npm test`:

```
{
  "name": "password",
  "version": "2.0.0",
  "description": "",
  "main": "index.html",
  "scripts": {
    "test": "jest",
    "test-watch": "jest --watch",
    "build-watch": "./node_modules/.bin/webpack -w",
    "build": "./node_modules/.bin/webpack"
  },
  "author": "Azat Mardan",
  "license": "MIT",
  "babel": {
    "presets": [
```

```
      "react"
    ]
  },
  "devDependencies": {
    "babel-core": "6.10.4",
    "babel-jest": "13.2.2",
    "babel-loader": "6.4.1",
    "babel-preset-react": "6.5.0",
    "jest-cli": "19.0.2",
    "react": "15.5.4",
    "react-dom": "15.5.4",
    "react-test-renderer": "15.5.4",
    "webpack": "2.4.1"
  }
}
```

А теперь рассмотрим другой способ рендера элементов React.

16.4.3. Неглубокий рендеринг

Иногда требуется протестировать один уровень рендера, то есть результат `render()` для компонента без рендера его дочерних элементов (если они есть). Такой подход упрощает тестирование, потому что он не требует наличия DOM: система создает элемент, и вы можете проверять различные тестовые утверждения относительно этого элемента. Прежде всего необходимо иметь пакет с именем `react-test-renderer` версии 15.5.4 (в старых версиях React этот класс был частью TestUtils, но в 15.5.4 он был исключен):

```
npm i react-test-renderer -SE
```

Ниже продемонстрировано тестирование того же элемента методом неглубокого рендеринга. Этот код может размещаться в том же тестовом файле ch16/password/__tests__/password.test.js. Для этого вы создаете специальный объект ререндера, а затем передаете ему компонент для получения неглубокого рендеринга:

```
const { createRenderer } = require('react-test-renderer/shallow')
const passwordRenderer = createRenderer()
passwordRenderer.render(<Password/>)
let p = passwordRenderer.getRenderOutput()
expect(p.type).toBe('div')
expect(p.props.children.length).toBe(6)
```

Выполняет неглубокий рендеринг

Применяет тестовое утверждение к результатам неглубокого рендеринга

Если теперь вывести `p` на консоль (например, `console.log(p)`), результат содержит дочерние элементы, но объект `p` не является экземпляром React. Рассмотрим результат неглубокого рендеринга:

```
{ '$$typeof': Symbol(react.element),
  type: 'div',
  key: null,
```

```
  ref: null,
  props:
    { className: 'well form-group col-md-6',
      children: [ [Object], [Object], [Object], [Object],
      [Object], [Object] ] },
  _owner: null,
  _store: {} }
```

Сравните с результатом `renderIntoDocument(<Password/>)`, который производит экземпляр элемента React `Password` с состоянием. Взгляните на результат рендеринга (обычного, не неглубокого):

```
Password {
  props: {},
  context: {},
  refs: {},
  updater:
    {...
      },
  state: { strength: {}, password: '',
    visible: false, ok: false },
    generate: [Function: bound generate],
    checkStrength: [Function: bound checkStrength],
    toggleVisibility: [Function: bound toggleVisibility],
    _reactInternalInstance:
      { _currentElement:
        { '$$typeof': Symbol(react.element),
          type: [Function: Password],
          key: null,
          ref: null,
          props: {},
          _owner: null,
          _store: {} },
          ...
        }
      }
```

Вы получаете состояние, недоступное при неглубоком рендеринге

Вы получаете элемент, который выглядит как результат неглубокого рендеринга

Не стоит и говорить, что тестирование поведения пользователя и вложенных элементов при неглубоком рендеринге невозможно. Однако такой рендеринг может использоваться для тестирования первого уровня дочерних элементов в компоненте, а также типа компонента. Эта возможность может использоваться для нестандартных классов компонентов.

В реальном мире неглубокий рендеринг может использоваться для узкоспециализированного (почти модульного) тестирования отдельного компонента и его рендера. Оно применяется тогда, когда нет необходимости тестировать дочерние элементы, поведение пользователя или изменяемые состояния компонента — другими словами, когда нужно протестировать только функцию `render()` одного элемента. На практике стоит начать с неглубокого рендеринга, а затем, если этого окажется недостаточно, продолжить с обычным.

Стандартные классы HTML могут анализировать и проверять `el.props`, так что в неглубоком рендеринге нет необходимости. Например, вот как можно создать якорный элемент и проверить его имя класса и тега на соответствие ожиданиям:

```
let el = <a className='btn'/>
expect(el.props.className).toBe('btn')
expect(el.type).toBe('a')
```

16.5. TestUtils: заключение

В этой главе вы многое узнали о TestUtils и Jest — достаточно, чтобы начать использовать их в ваших собственных проектах. Именно этим мы займемся в части 2 книги: применением Jest и TestUtils для разработки через реализацию поведения (BDD) компонентов React (главы 18–20). Виджет генерирования паролей рассматривается в главе 19, если вы захотите взглянуть на настройку Webpack и все зависимости, используемые в реальном мире.

За дополнительной информацией о TestUtils обращайтесь к официальной документации по адресу https://facebook.github.io/react/docs/test-utils.html. Jest — весьма обширная тема, и полное ее описание выходит за рамки книги. За дополнительной информацией обращайтесь к официальной документации API: https://facebook.github.io/jest/docs/api.html#content.

Наконец, библиотека Enzyme (https://github.com/airbnb/enzyme, http://mng.bz/Uy4H) предоставляет расширенную функциональность и набор методов по сравнению с TestUtils, а также более компактные имена методов. Она была разработана в AirBnb и требует наличия TestUtils, а также библиотеки `jsdom` (которая поставляется с Jest, так что `jsdom` понадобится вам только в том случае, если вы не используете Jest).

Тестирование не для слабаков. Некоторых разработчиков оно настолько пугает, что они пропускают его — но не вы. Вы дочитали эту главу до конца. Поздравляю! Ваш код будет более качественным, ваша разработка пойдет быстрее и станет более приятной. Вам не придется вскакивать по ночам и исправлять неработающий сервер — ну или по крайней мере вы это будете делать так часто, как разработчики, которые не пишут тесты.

16.6. Вопросы

1. В какой из следующих папок должны находиться тесты Jest: tests, __test__ или __tests__?
2. TestUtils устанавливается npm из `react-addons-test-utils`. Да или нет?
3. Какой метод TestUtils позволяет найти один компонент по классу HTML?

4. Какое выражение `expect` используется для сравнения объектов (глубокое сравнение)?
5. Какой метод используется для тестирования поведения при навигации указателя мыши: `TestUtils.Simulate.mouseOver(node)`, `TestUtils.Simulate.onMouseOver(node)` или `TestUtils.Simulate.mouseDown(node)`?

16.7. Итоги

- Чтобы установить Jest, используйте команду `npm i jest-cli --save-dev`.
- Чтобы протестировать модуль, отключите для него автоматическую имитацию зависимостей вызовом `jest.dontMock()`.
- Используйте `expect.toBe()` и другие функции Expect.
- Чтобы установить TestUtils, используйте команду `npm i react-addons-test-utils --save-dev`.
- Используйте метод TestUtils.Simulate.eventName(node), где eventName — событие React (без префикса `on`), для тестирования событий DOM.
- Используйте методы `scry...` для получения нескольких элементов.
- Используйте методы `find...` для получения одного элемента (при наличии нескольких элементов будет получена ошибка: *«Не было найдено ровно одно совпадение (найдено: 2+)»*).

16.8. Ответы

1. `__tests__` по соглашению, используемому в Jest.
2. Да. TestUtils — отдельный модуль npm.
3. `findRenderedDOMComponentWithClass()`.
4. `expect(OBJECT).toEqual(value)` сравнивает объекты без проверки их тождественности (которая выполняется оператором `===` или `toBe()`).
5. `TestUtils.Simulate.mouseOver(node)`. Событие `mouseOver` инициируется при наведении указателя мыши.

17 Использование React с Node и универсальный JavaScript

Посмотрите вступительный видеоролик к этой главе, отсканировав QR-код или перейдя по ссылке http://reactquickly.co/videos/ch17.

ЭТА ГЛАВА ОХВАТЫВАЕТ СЛЕДУЮЩИЕ ТЕМЫ:

- Использование React на сервере.
- Универсальный JavaScript.
- Использование React с Node.
- Работа с React и Express.
- Использование универсального JavaScript с Express и React.

React в первую очередь является библиотекой клиентской части для построения полнофункциональных одностраничных приложений или простых пользовательских интерфейсов в браузере. Тогда зачем возиться с использованием React на сервере? Разве генерирование HTML на сервере не считается старомодным? Да и нет. Оказывается, при построении веб-приложений, которые *всегда* выполняют рендеринг в браузере, вы упускаете ряд ключевых возможностей, например возможность высокого ранжирования в результатах поиска Google, а возможно, даже потерю миллионных прибылей.

Хотите узнать почему? Читайте дальше. Эту главу можно пропустить только в одном случае: если вас не интересует быстродействие ваших приложений (то есть если вы новичок в области разработки). Все остальные могут продолжить читать. Вы найдете здесь неоценимые знания, которые помогут вам в построении современных приложений, а заодно помогут произвести впечатление на коллег, когда

вы упомянете термин «универсальный JavaScript» на ежедневном совещании. Вы также научитесь использовать React с Node и строить серверы Node, а к концу главы поймете, как строить универсальные приложения JavaScript с React.js и Express.js (самый популярный фреймворк Node.js).

СОВЕТ Если вы еще не сталкивались с Express, ознакомьтесь с моей книгой «Pro Express.js» (Apress, 2014), в которой рассматривается текущая версия v4; книга подробная и до сих пор актуальная. Также см. «Express in Action» Эвана Хана (Evan Hahn) (Manning, 2015). Также ознакомьтесь с сетевым учебным курсом Express Foundation: https://node.university/p/express-foundation. Если вы знакомы с Express, но хотите освежить память, краткая сводка Express.js приведена в приложении B, а установка Express рассматривается в приложении A.

ПРИМЕЧАНИЕ Исходный код примеров этой главы доступен по адресам www.manning.com/books/react-quickly и https://github.com/azat-co/react-quickly/tree/master/ch17. Также некоторые демонстрационные примеры доступны по адресу http://reactquickly.co/demos.

17.1. Зачем нужен React на сервере? И что такое универсальный JavaScript?

Возможно, вы уже слышали об универсальном JavaScript в связи с веб-разработкой. Этот термин стал настолько модным, что на каждой конференции по веб-технологиям в 2016 году по этой теме проводилось как минимум несколько докладов. У термина «универсальный JavaScript» даже есть несколько синонимов, например: «изоморфный JavaScript» и «JavaScript полного стека». Для простоты в этой главе я буду придерживаться термина «универсальный». Этот раздел поможет вам понять, для чего нужен изоморфный/универсальный JavaScript.

Но прежде чем давать определение универсального JavaScript, обсудим некоторые проблемы, с которыми вы столкнетесь при построении одностраничных приложений. Три главные проблемы:

- *Отсутствие поисковой оптимизации* (SEO) — одностраничные приложения (SPA) генерируют HTML исключительно в браузере, а поисковым ботам это не подходит.
- *Плохое быстродействие* — огромные упакованные файлы и вызовы AJAX ухудшают быстродействие (особенно при загрузке первой страницы, когда оно критично).
- *Трудности с сопровождением* — часто SPA-приложения приводят к дублированию кода в браузере и на сервере.

А теперь поближе рассмотрим каждую из этих проблем.

17.1.1. Корректное индексирование страниц

SPA-приложения, построенные на базе таких фреймворков, как Backbone.js, Angular.js, Ember.js и др., широко применяются для защищенных приложений, то есть приложений, требующих ввода имени пользователя и пароля для получения доступа (например, см. рис. 17.1). Многие SPA-приложения предоставляют защищенные ресурсы и не нуждаются в индексировании, но подавляющее большинство веб-сайтов не защищается входными данными.

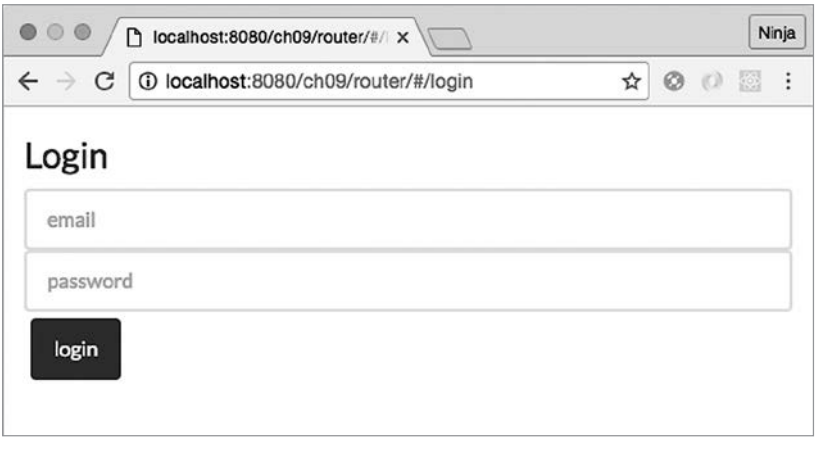


Рис. 17.1. SPA-приложение не нуждается в поддержке SEO, потому что оно находится за экраном входа

Для таких общедоступных приложений поддержка SEO важна и практически обязательна, потому что их коммерческая сторона сильно зависит от оптимизации поиска и естественного трафика. Большинство веб-сайтов относится к этой категории.

К сожалению, при использовании архитектуры SPA для общедоступных приложений, которые должны иметь нормальную поддержку индексирования поисковыми системами, сделать это не так просто. SPA-приложения зависят от рендеринга в браузере, поэтому вам придется либо заново реализовать шаблоны на сервере, либо заранее сгенерировать статические страницы HTML с использованием консольных браузеров только для ботов поисковых систем.

ПОДДЕРЖКА РЕНДЕРИНГА В БРАУЗЕРЕ СО СТОРОНЫ GOOGLE

Недавно компания Google наделила своих поисковых ботов поддержкой генерирования JavaScript. Казалось бы, это означает, что разметка HTML, сгенерированная браузером, теперь будет индексироваться правильно. Можно подумать, что при использовании Angular с сервером REST API рендеринг на стороне сервера не нужен. К сожалению, это не так.

> Следующий фрагмент взят из сообщения «Understanding Web Pages Better» (http://mng.bz/Yv3B) в Google Webmaster Central Blog: «Иногда в процессе рендеринга не все проходит идеально, что может отрицательно сказаться на результатах поиска для вашего сайта». Суть в том, что компания Google не уговаривает нас положиться на ее механизм индексирования SPA. Google не может гарантировать, что содержимое кэша, индекса и результатов поиска в точности соответствует тому, что сгенерирует ваше SPA-приложение. Итак, для пущей надежности необходимо, чтобы генерируемая без JavaScript разметка была как можно ближе к разметке с включенным JavaScript.

С универсальным JavaScript и React в частности можно генерировать на сервере разметку HTML для ботов из тех же компонентов, которые используются браузерами для генерирования разметки для пользователей. Отпадает необходимость в неудобных консольных браузерах для генерирования HTML на сервере. Беспроигрышная ситуация!

17.1.2. Повышение быстродействия с ускорением загрузки

Хотя некоторые приложения должны обеспечивать правильное индексирование поисковых систем, другие ориентируются на максимальное быстродействие. Исследования, проведенные такими сайтами, как http://mobile.walmart.com[1] и http://twitter.com[2], показали, что для повышения быстродействия первая страница (первая загрузка) должна рендериться на сервере. Компании теряют миллионы долларов из-за того, что пользователи уходят, потому что первая страница загружается недостаточно быстро.

Веб-разработчику, который живет и работает с хорошим подключением к интернету, легко забыть, что пользователи могут приходить на его сайт по медленному подключению. То, что у вас загружается за долю секунды, в других случаях может загружаться по полминуты. Внезапно пакет размером более 1 Мбайт становится слишком большим. А ведь загрузка упакованного файла — всего лишь полдела: SPA-приложение должно выдавать AJAX-запросы к серверу для загрузки данных, пока пользователь терпеливо разглядывает индикатор Loading... Как же, ждите — одни пользователи уже ушли, а другие начинают злиться.

Пользователь должен как можно быстрее увидеть функциональную веб-страницу, не просто заготовку из HTML с надписью Loading... Другой код может быть загружен позднее, пока пользователь просматривает веб-страницу.

---

[1] Kevin Decker, «Mobile Server Side Rendering», GitHub Gist, 2014, http://mng.bz/2B6P.

[2] Dan Webb, «Improving Performance on twitter.com», Twitter, 29 мая 2012 г., http://mng.bz/2st9.

С универсальным JavaScript вы можете легко сгенерировать HTML для вывода первой страницы на сервере. В результате во время загрузки первой страницы пользователь не видит раздражающего сообщения Loading... Разметка HTML содержит реальные данные. Пользователь видит функциональную страницу, а это улучшает его впечатления от сайта.

Прирост быстродействия обусловлен тем фактом, что пользователям не нужно дожидаться разрешения вызовов AJAX. Также существуют и другие средства оптимизации быстродействия — например, предварительная загрузка данных и кэширование их на сервере до поступления запросов AJAX (именно так мы действовали в DocuSign при реализации маршрутизации данных[1]).

17.1.3 Упрощенное сопровождение кода

Код — это пассив. Чем больше у вас кода, тем больше усилий вам и вашей команде потребуется для его сопровождения. По этим причинам нужно избегать использования разных шаблонов и логики для одних и тех же страниц. Избегайте дублирования. Не повторяйтесь (DRY[2]).

К счастью, платформа Node.js, являющаяся важнейшей частью универсального JavaScript, позволяет без особых усилий использовать клиентские/браузерные модули на сервере. Многие шаблонизаторы, включая Handlebars.js, Mustache, Dust.js и др., могут использоваться на сервере.

Итак, если вы знаете возникающие проблемы и то, как универсальный JavaScript способствует их решению, какое практическое решение можно предложить?

17.1.4. Универсальный JavaScript с React и Node

«Универсальность» в контексте веб-разработки означает, что один и тот же код (обычно написанный на JavaScript) может использоваться как на стороне сервера, так и на стороне клиента. Один из узкоспециализированных случаев применения универсального JavaScript — выполнение рендеринга на сервере и в клиенте из одного исходного кода. Универсальный JavaScript часто подразумевает использование JavaScript и Node.js, потому что этот язык и комбинация платформ обеспечивают возможность повторного использования библиотек.

Браузерный код JavaScript может выполняться в среде Node.js с некоторыми изменениями. Вследствие такой взаимозаменяемости экосистема Node.js и JavaScript содержит широкий выбор изоморфных фреймворков, таких как React.js (http://facebook.github.io/react), Next.js (https://github.com/zeit/next.js), Catberry (http://catberry.org/), LazoJS (https://github.com/lazojs/lazo), Rendr (https://github.com/rendrjs/rendr), Meteor (https://meteor.com) и многие другие. На рис. 17.2 показано, как работает

[1] Ben Buckman, «The New DocuSign Experience, All in Javascript», DocuSign Dev, 30 марта 2014 г., http://mng.bz/4773.

[2] Https://ru.wikipedia.org/wiki/Don't_repeat_yourself.

универсальный/изоморфный стек: изоморфный код совместно используется на сервере и в клиенте.

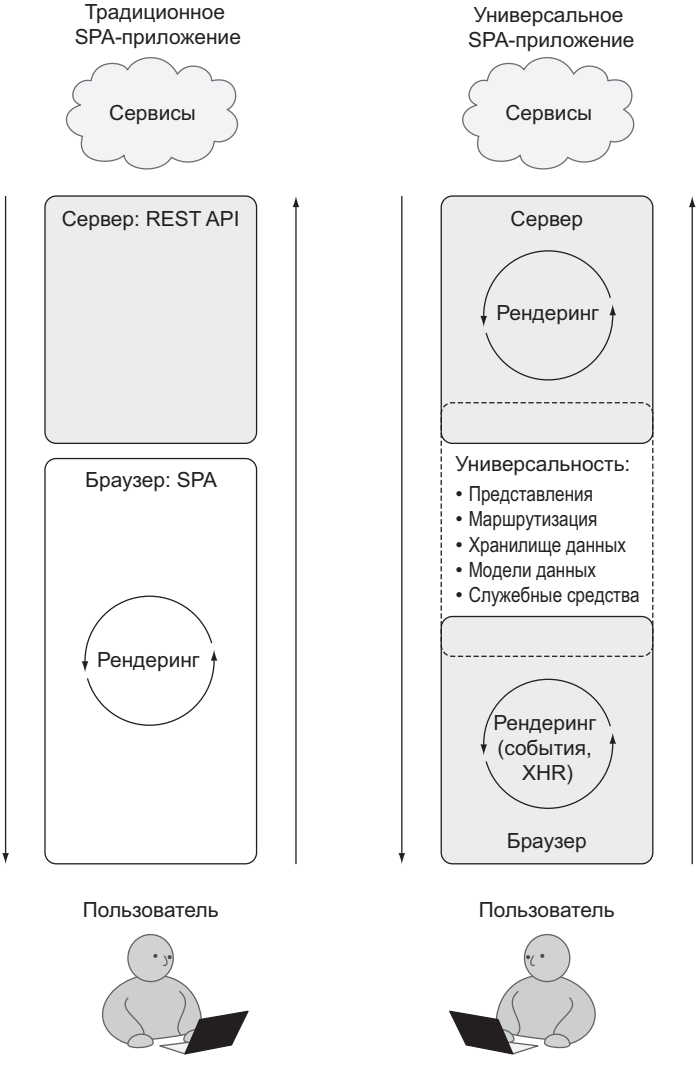


Рис. 17.2. Универсальное генерирование HTML и совместное использование кода между браузером и сервером, в отличие от подхода традиционных SPA-приложений

В реальном приложении архитектура универсального JavaScript состоит из следующих частей:

- Код React на стороне клиента для браузера. Это может быть SPA-приложение или простой пользовательский интерфейс, выдающий запросы AJAX.

- Сервер Node.js, генерирующий HTML для первой страницы на сервере и предоставляющий браузерный код React с теми же данными. Эта часть может быть реализована с использованием Express и либо шаблонизатора, либо компонентами React в качестве шаблонизатора.
- Webpack для компиляции JSX для сервера и браузера.

Модель изображена на рис. 17.3.

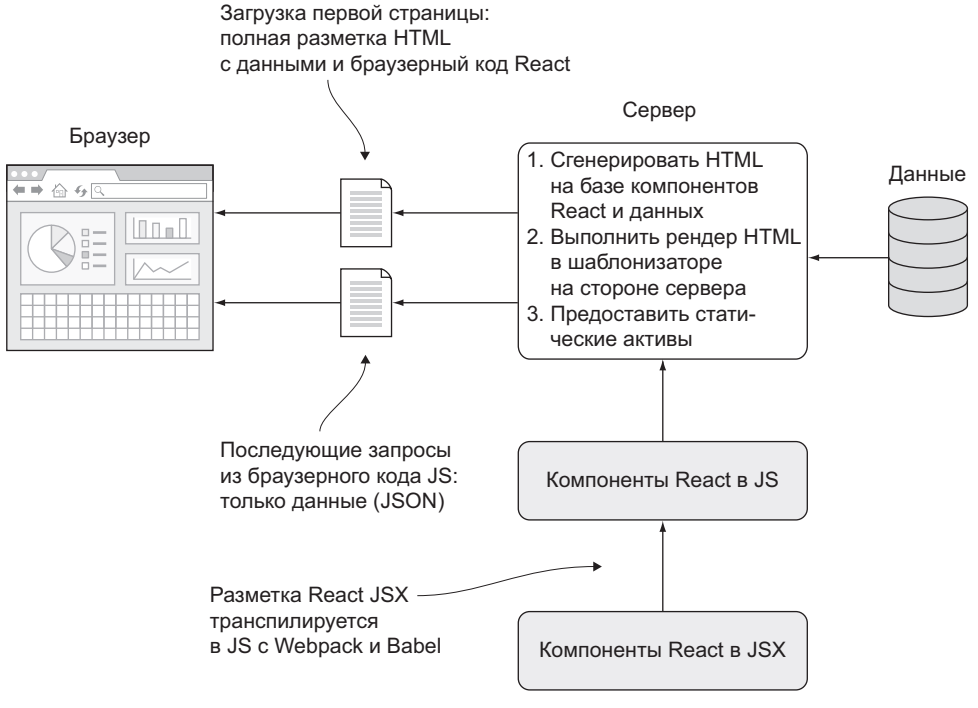


Рис. 17.3. Практическое применение универсального JavaScript с React, Node и Express

Возможно, вы думаете: «Так покажите мне этот замечательный универсальный JavaScript!» Ладно, рассмотрим практический пример рендера компонентов React на сервере. Мы сделаем это постепенно, так как в использовании паттерна универсального JavaScript задействовано несколько компонентов (составляющих, а не компонентов React). Вам нужно научиться решать следующие задачи:

- *Генерировать HTML из компонентов React* — на входе компоненты React, на выходе простая разметка HTML; никаких серверов HTTP(S) еще нет.
- *Выполнять рендер кода HTML, сгенерированного из компонентов React на серверах Express*, — по аналогии с предыдущим пунктом, но теперь React используется в шаблонизаторе для 100 % рендера на стороне сервера (на стороне браузера React пока не используется).

- *Реализовать и предоставлять браузерные файлы React через Express* — через какое-то время вам потребуется сервер HTTP(S), Express — один из вариантов. До сих пор мы использовали node-static или Webpack Dev Server. На этой стадии на стороне сервера разметка HTML уже не генерируется, предоставляются только построенные/откомпилированные статические ассеты.

В итоге вы будете использовать React для генерирования серверной разметки HTML, одновременно загружая браузерный код React — священный Грааль универсального JavaScript. Но прежде чем летать, нужно научиться ходить!

17.2. React с Node

Начнем с базового случая: генерирования разметки HTML из сценария Node. В этом примере нет ни серверов, ни других сложностей — только импортирование компонентов и генерирование HTML. Убедитесь в том, что вы используете Node версии не ниже 6-й, а ваша версия npm не ниже 3-й.

Чтобы генерировать HTML из компонентов React на сервере, необходимо изучить небольшой набор методов. Для начала вам понадобятся модули npm `react` и `react-dom`. Инструкции по установке React и npm приведены в приложении А. В этом примере используется React и React DOM версии 15.

Если вы еще не писали код Node на стороне сервера, может возникнуть вопрос: где размещается весь этот серверный код? Он размещается в простом текстовом файле; присвойте ему имя index.js. Компонент React находится в файле email.js (пока я ограничиваюсь простым кодом JavaScript без JSX). Эти два файла должны находиться в одной папке (ch17/node).

Структура проекта выглядит так:

```
/node
  /node_modules  ◄—— Зависимости
  email.js
  email.jsx  ◄—— Компонент адреса электронной почты
  index.js  ◄—— Код Node
  package.json
```

Сначала включите модули в своем коде на стороне сервера в файл node/index.js.

Листинг 17.1. Подготовительный код на стороне сервера

```
const ReactDOMServer = require('react-dom/server') ◄—— Импортирует класс ReactDOMServer
const React = require('react')
const Email = React.createFactory(require('./email.js'))  ◄——
...
                                    Создает функцию, которая возвращает
                                                элементы класса Email
```

К чему здесь `createFactory()`? После импортирования email.js вы получаете класс компонента; но нужен вам элемент React. А значит, вы можете использовать JSX: `createElement()` или `createFactory()`. Последний вызов дает функцию, которая при вызове вернет элемент.

После импортирования компонентов выполните метод `renderToString()` для `ReactDOMServer`:

```
const emailString = ReactDOMServer.renderToString(Email())
```

Фрагмент кода из index.js:

```
const ReactDOMServer = require('react-dom/server')
const React = require('react')
const Email = React.createFactory(require('./email.js'))

const emailString = ReactDOMServer.renderToString(Email())
console.log(emailString)
// ...
```

ИМПОРТИРОВАНИЕ JSX

Другой подход к использованию JSX — преобразование «на ходу». Библиотека `babel-register` расширяет `require` так, чтобы вы могли настроить require один раз, а затем импортировать JSX так же, как любые другие файлы JS.

Чтобы импортировать JSX, используйте `babel-register` так, как показано здесь в файле `index.js`, а также установите `babel-register` и `babel-preset-react` (используйте для этого npm):

```
require('babel-register')({
  presets: [ 'react' ]
})
```

Содержит ли email.js обычный код JavaScript? В данном случае должен содержать. Вы можете «встроить» JSX в обычный код JS при помощи Webpack.

Листинг 17.2. Серверный компонент email (node/email.jsx)

```
const React = require('react')

const Email = (props)=> {
  return (
    <div>
      <h1>Thank you {(props.name) ? props.name: '' }
        for signing up!</h1>
      <p>If you have any questions, please contact support</p>
    </div>
  )
}

module.exports = Email
```

Вы будете получать строки, сгенерированные компонентами React. Эти строки можно использовать в вашем любимом шаблонизаторе для вывода на веб-странице или где-то еще (например, в электронной почте в формате HTML). В моем случае файл email.js (ch17/node/email.js) с заголовком и абзацем генерирует следующие строки HTML с атрибутами универсального кода React.

Листинг 17.3. Строки, сгенерированные node/email.js

```
<div data-reactroot="" data-reactid="1" data-react-checksum="1319067066">
  <h1 data-reactid="2">
    <!-- react-text: 3 -->Thank you <!-- /react-text -->
    <!-- react-text: 4 -->
    <!-- /react-text -->
    <!-- react-text: 5 -->for signing up!<!-- /react-text -->
  </h1>
  <p data-reactid="6">If you have any questions, please contact support</p>
</div>
```

Откуда взялись атрибуты `data-reactroot`, `data-reactid` и `data-react-checksum`? Вы их не вставляли, это делает React. Почему? Для React в браузере и универсального JavaScript (об этом в следующем разделе).

Если вам не нужна разметка React, необходимая браузерному коду React (например, если вы создаете сообщение электронной почты в формате HTML), используйте метод `ReactDOMServer.renderToStaticMarkup()`. Он работает аналогично `renderToString()`, но удаляет атрибуты `data-reactroot`, `data-reactid` и `data-react-checksum`. В этом случае React работает просто как любой другой шаблонизатор.

Например, вы можете загрузить компонент из email.js и сгенерировать HTML-методом `renderToStaticMarkup()` вместо `renderToString()`:

```
const emailStaticMarkup = ReactDOMServer.renderToStaticMarkup(Email())
```

В полученном значении `emailStaticMarkup` нет атрибутов React:

```
<div><h1>Thank you for signing up!</h1><p>If you have any questions,
➥ please contact support</p></div>
```

Хотя для электронной почты браузерный код React не нужен, для архитектуры универсального JavaScript с React используется исходный метод `renderToString()`. React на стороне сервера добавляет в HTML секретные ингредиенты — контрольные суммы (атрибуты HTML `data-react-checksum`). React в браузере сравнивает эти контрольные суммы, и если они совпадают, браузерные компоненты не выполняют излишнего генерирования/рендеринга/обновления изображения. Тем самым предотвращается мерцание контента (которое часто возникает из-за повторного рендера). Контрольные суммы совпадают, если данные, переданные компонентам на стороне сервера, в точности совпадают с данными в браузере. Но как передать данные компонентам, созданным на сервере? В свойствах!

Если вам потребуется передать свойства, передайте их в объектных параметрах. Скажем, компоненту `Email` можно передать имя (`Johny Pineappleseed`):

```
const emailStringWithName = ReactDOMServer.renderToString(Email({
  name: 'Johny Pineappleseed'
}))
```

Ниже приведен полный код ch17/node/index.js с тремя способами генерирования HTML — статическим, строковым и строковым со свойством:

```
const ReactDOMServer = require('react-dom/server')
const React = require('react')
const Email = React.createFactory(require('./email.js'))

const emailString = ReactDOMServer.renderToString(Email())
const emailStaticMarkup = ReactDOMServer.renderToStaticMarkup(Email())
console.log(emailString)
console.log(emailStaticMarkup)

const emailStringWithName =
➥ ReactDOMServer.renderToString(Email({name: 'Johny Pineappleseed'}))
console.log(emailStringWithName)
```

Так выполняется генерирование HTML компонентами React в простом коде Node — без серверов и прочих сложностей. А теперь посмотрим, как использовать React в сервере Express.

17.3. React и Express: генерирование кода из компонентов на стороне сервера

Express.js — один из самых популярных фреймворков Node.js, возможно, *самый* популярный. Он прост, но обладает широкими возможностями настройки. Существуют сотни плагинов, называемых *промежуточными модулями* (middleware), которые могут использоваться с Express.js.

В высокоуровневом представлении технологического стека Express и Node занимают место сервера HTTP(S), фактически заменяя такие технологии, как Microsoft IIS (www.iis.net), Apache httpd (https://httpd.apache.org), nginx (www.nginx.com) и Apache Tomcat (http://tomcat.apache.org). Уникальность Express и Node заключается в том, что они позволяют строить масштабируемые высокопроизводительные системы благодаря неблокирующей природе ввода/вывода в Node (http://github.com/azat-co/you-dont-know-node). Главные достоинства Express — гигантская экосистема промежуточных модулей и проверенная временем, стабильная кодовая база.

К сожалению, подробное описание фреймворка выходит за рамки книги, но мы создадим маленькое приложение Express и встроим в него React. Это ни в коем случае не может считаться глубоким знакомством с Express.js, но вы по крайней

мере познакомитесь с самым популярным веб-фреймворком Node.js. Считайте это своего рода кратким курсом Express.

СОВЕТ Как упоминалось ранее, в приложении A рассказано, как установить node.js и Express.

17.3.1. Рендеринг простого текста на стороне сервера

Построим серверы HTTP и HTTPS с использованием Express, а затем сгенерируем HTML на стороне сервера с использованием React, как схематично показано на рис. 17.4. Простейший сценарий использования React в Express как шаблонизатора заключается в генерировании строки HTML без разметки (контрольных сумм) и отправки ее как ответа на запрос. В листинге 17.4 представлена страница `/about`, отрендеренная из компонента React about.js.

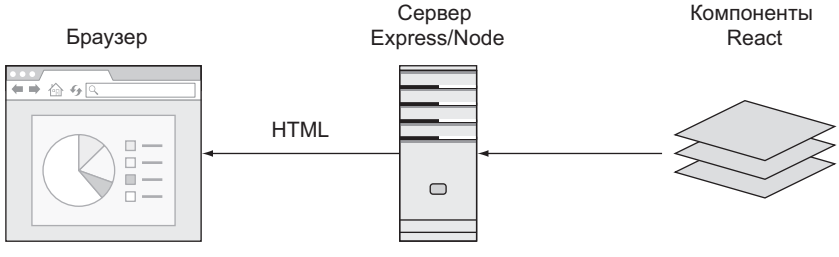


Рис. 17.4. Сервер Express/Node генерирует разметку HTML и отправляет ее браузеру

Листинг 17.4. Использование React с Express для вывода HTML на странице

```
const express = require('express')  ◄── Импортирует библиотеку express
const app = express()
const http = require('http')

const ReactDOMServer = require('react-dom/server')
const React = require('react')
const About =
  React.createFactory(require('./components/about.js'))  ◄── Импортирует компонент About и создает объект React

app.get('/about', (req, res, next) => {
  const aboutHTML = ReactDOMServer.renderToStaticMarkup(About())
  response.send(aboutHTML)  ◄── Отправляет строку HTML обратно клиенту в ответе на запрос /about
})

http.createServer(app)  ◄── Создает экземпляр сервера HTTP и запускает его
  .listen(3000)
```

Такое решение будет работать, но `/about` не может считаться полноценной страницей без `<head>` и `<body>`. Лучше использовать правильный шаблонизатор (например, Handlebars) для структуры и элементов HTML верхнего уровня. Возможно, вас

также интересует, что такое `app.get()` и `app.listen()`? Рассмотрим другой пример, и все станет ясно.

17.3.2. Рендеринг страницы HTML

В этом разделе рассматривается более интересный пример, в котором используются внешние плагины и шаблонизатор. Идея приложения остается прежней: предоставление разметки HTML, сгенерированной из React с использованием Express. Страница выводит текст, сгенерированный из about.jsx (рис. 17.5). Ничего выдающегося, но зато просто, а начинать лучше с простого.

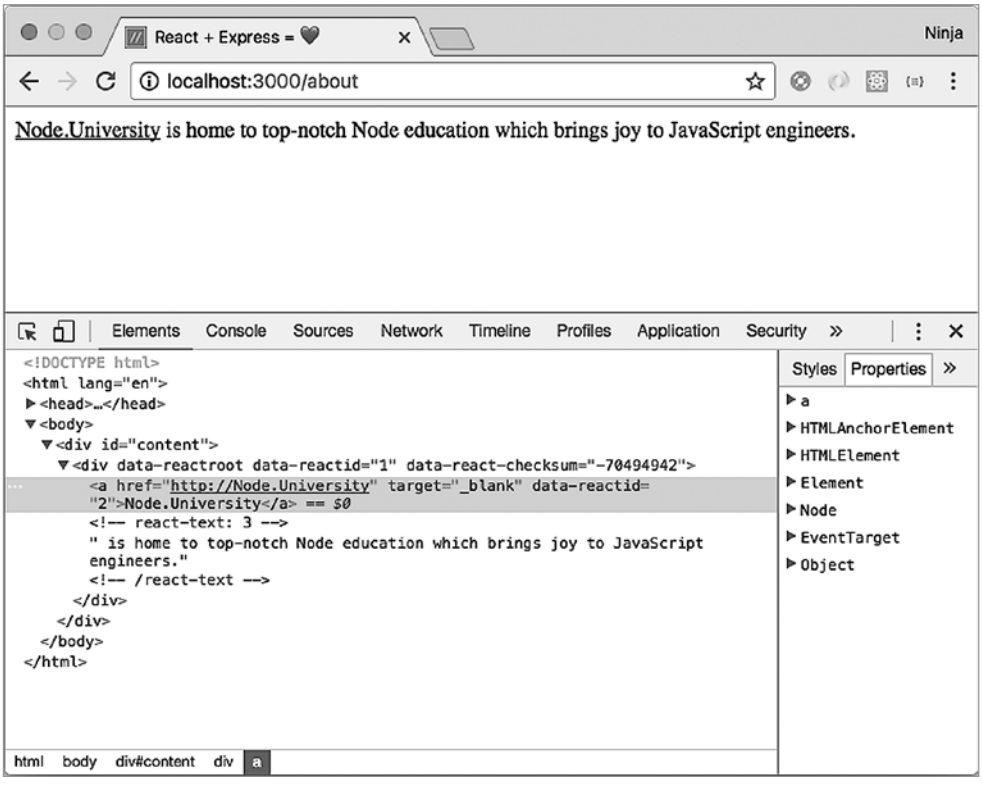


Рис. 17.5. Рендеринг разметки из компонента React на стороне сервера

Создайте папку с именем react-express. (Пример находится в ch17/react-express.) Итоговая структура проекта выглядит так:

```
/react-express
  /components
    about.jsx
```

```
  /views
    about.hbs
  index.js
  package.json
```

Создайте файл package.json командой `npm init -y`, а затем установите Express из npm следующей командой:

```
$ npm install express@4.14.0 --save
```

Как и с любым приложением Node, откройте редактор и создайте файл. Как правило, создается серверный файл с именем index.js, app.js или server.js, который позднее будет запущен командой node. В данном случае присвойте ему имя index.js.

Файл состоит из следующих частей:

- *Импортирование* — включение зависимостей (таких, как Express и его плагины).
- *Конфигурации* — некоторые параметры конфигурации (например, используемый шаблонизатор).
- *Промежуточные модули* — определение распространенных действий, выполняемых для всех входящих запросов, таких как проверка данных, аутентификация, сжатие и т. д.
- *Маршруты* — определение URL-адресов, обрабатываемых сервером, таких как `/accounts`, `/users` и т. д., а также их действий.
- *Обработчики ошибок* — вывод содержательных сообщений или веб-страниц при возникновении ошибок.
- *Запуск* — запуск сервера(-ов) HTTP и/или HTTPS.

Структура серверного файла Node и Express выглядит так:

```
const express = require('express')  <-- Импортирует модули
const app = express()
const errorHandler = require('errorhandler')
const http = require('http')
const https = require('https')
// Импортирование других модулей
// ...

app.set('view engine', 'hbs')  <-- Задает конфигурацию

app.get('/',  <-- Определяет маршруты (в этом фрагменте нет «чистого» промежуточного кода)
  // ...
)

app.get('/about',
  // ...
)
```

```
// ...

app.use(errorHandler) ◄—— Определяет обработчики ошибок (разновидность промежуточных модулей)

http.createServer(app)  ◄——  Запускает сервер HTTP
  .listen(3000)

// ...
if (typeof options != 'undefined')
  https.createServer(app, options)  ◄——  Запускает сервер HTTPS
    .listen(443)
```

Теперь заглянем поглубже. Секция импортирования тривиальна: в ней включаются зависимости и создаются экземпляры объектов. Например, чтобы импортировать фреймворк Express.js и создать его экземпляр, включите следующие строки:

```
var express = require('express')
var app = express()
```

Конфигурация

Конфигурации задаются вызовом `app.set()`, в котором первый аргумент содержит строку, а второй — значение. Например, чтобы выбрать шаблонизатор hbs (www.npmjs.com/package/hbs), используйте конфигурацию `view engine`:

```
app.set('view engine', 'hbs')
```

hbs — шаблонизатор Express для языка шаблонов Handlebars (http://handlebarsjs.com). Возможно, вы работали с Handlebars или с одним из его близких родственников — Mustache, Blaze и т. д. Ember также использует Handlebars (http://mng.bz/90Q2). Это распространенный, легкий для освоения шаблон, именно поэтому мы воспользуемся им.

> **ПРЕДУПРЕЖДЕНИЕ** Чтобы Express правильно использовал шаблонизатор, необходимо установить пакет hbs. Выполните команду `npm i hbs --save`.

Промежуточные модули

В следующем разделе настраиваются промежуточные модули. Например, чтобы приложение предоставляло статические активы, используйте промежуточный модуль `static`:

```
app.use(express.static(path.join(__dirname, 'public')))
```

Промежуточный модуль `static` очень удобен, потому что он превращает Express в статический сервер HTTP(S), обрабатывающий запросы к заданной папке (`public` в данном примере), как бы это делал NGINX или Apache httpd.

Маршруты

Затем идут маршруты, также называемые «конечными точками», «ресурсами», «страницами» и многими другими терминами. Вы определяете шаблон URL-адреса, который будет применяться Express к реальным URL-адресам входных запросов. При обнаружении совпадения Express применяет логику, связанную с этим URL-адресом; это называется *обработкой* запроса. Обработка может включать все что угодно, от отображения статической разметки HTML для страницы «404 Not Found» до передачи запроса другому сервису и кэшированию ответа перед его отправкой клиенту.

Маршруты являются важной частью веб-приложения, потому что они определеляют маршрутизацию URL и в некотором смысле выполняют функции контроллеров в старом добром паттерне MVC (модель-представление-контроллер). В Express для определения маршрутов используется паттерн `app.NAME()`, где `NAME` — имя команды (метода) HTTP в нижнем регистре. Например, синтаксис GET-запроса конечной точки `/` (домашняя страница или пустой URL-адрес), возвращающего строку «Hello», выглядит так:

```
app.get('/', (request, response, next) => {
  response.send('Hello!')
})
```

Для маршрута/страницы `/about` можно изменить первый аргумент (шаблон URL), а также сгенерировать строку HTML:

```
app.get('/about', (req, res, next) => {
  response.send(`<div>
  <a href="https://node.university" target="_blank">Node.University</a>
    is home to top-
      notch Node education which brings joy to JavaScript engineers.
</div>`)
})
```

Построение структуры с Handlebars

Теперь нужно отрендерить разметку HTML из шаблона Handlebars, потому что Handlebars предоставляет общую структуру, включающую такие структурные элементы, как `<html>` и `<body>`. Другими словами, вы используете React для UI-элементов и Handlebars для структуры.

Создайте новую папку views, в которой находится шаблон about.hbs:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <title>React + Express = ♥ </title>  ◄——  Использует сердечко ♥
```

```
    <meta name="author" content="Azat" />
  </head>

  <body>
    <div id="content">{{{about}}}</div>
  </body>
</html>
```

Использует тройные фигурные скобки для вывода неэкранированного HTML из переменной about (передаваемой в index.js)

Рендеринг страницы

В маршруте (в файле ch17/react-express/index.js) замените `response.send()` на `response.render()`:

```
// ...
const React = require('react')
require('babel-register')({
  presets: [ 'react' ]
})
const About =
  React.createFactory(require('../components/about.jsx'))
// ...
app.get('/about', (request, response, next) => {
  const aboutHTML = ReactDOMServer.renderToString(About())
  response.render('about', {about: aboutHTML})
})
// ...
```

Расширяет require для преобразования JSX «на ходу», что позволяет импортировать/включать файлы JSX

Подготавливает компонент About

Генерирует HTML-строку React с разметкой React

Передает HTML-строку React шаблону Handlebars about.hbs

Маршруты Express могут генерировать данные по шаблонам Handlebars с такими данными, как строковая переменная `about`, или же отправлять ответ в строковом формате.

ОБЯЗАТЕЛЬНО ЛИ ИСПОЛЬЗОВАТЬ РАЗНЫЕ ШАБЛОНИЗАТОРЫ?

Вместо Handlebars также можно использовать React для структуры. Для этой цели существует библиотека `express-react-views` (www.npmjs.com/package/express-react-views). Это возможно только для статической разметки, не для React в браузере.

Здесь эта тема не рассматривается, потому что она требует интенсивного использования `dangerouslySetInnerHTML`[1], не поддерживает некоторые аспекты HTML и часто сбивает с толку начинающих разработчиков Express-React. По моему скромному мнению, применение React для структуры особой пользы не приносит.

[1] См. главу 3 или https://facebook.github.io/react/docs/dom-elements.html#dangerouslysetinnerhtml.

Обработка ошибок

Обработчики ошибок похожи на промежуточные модули. Например, они могут импортироваться из пакетов, таких как `errorhandler` (www.npmjs.org/package/errorhandler):

```
const errorHandler = require('errorhandler')
...
app.use(errorHandler)
```

Или их можно создать в index.js:

```
app.use((error, request, response, next) => {
  console.error(request.url, error)
  response.send('Wonderful, something went wrong...')
})
```

Чтобы запустить обработчик ошибки, вызовите `next(error)` в обработчике запроса или в промежуточном модуле. `error` — это объект ошибки, который создается вызовом `new Error('Ooops')`, где «Ooops» становится сообщением об ошибке. Пример из `/about`:

```
app.get('/about', (request, response, next) => {
  // ... аномальная ситуация
  let somethingWeirdHappened = true
  if (somethingWeirdHappened) return next(new Error('Ooops'))
})
```

Не забывайте использовать `return`. За дополнительной информацией об обработке ошибок в Node и Express обращайтесь к курсу «Node Patterns» (http://node.university/p/node-patterns) или моему сообщению «Node Patterns: From Callbacks to Observer» (http://webapplog.com/node-patterns).

Запуск сервера

Наконец, чтобы запустить приложение, выполните `listen()` с номером порта и обратным вызовом (не обязательно):

```
http.createServer(app).listen(portNumber, callback)
```

В нашем примере это выглядит так:

```
http.createServer(app)
  .listen(3000)
```

В листинге 17.5 приведен полный код сервера в ch17/react-express/index.js; убедитесь, что вы ничего не упустили.

Листинг 17.5. Полный код для React/Express/сервера hbs

```
const fs = require('fs')
const express = require('express')
const app = express()
const errorHandler = require('errorhandler')
const http = require('http')
const https = require('https')

const React = require('react')
require('babel-register')({
  presets: [ 'react' ]
})

const ReactDOMServer = require('react-dom/server')
const About = React.createFactory(require('./components/about.jsx'))

app.set('view engine', 'hbs')
app.get('/', (request, response, next)=>{
  response.send('Hello!')
})

app.get('/about', (request, response, next) => {
  const aboutHTML = ReactDOMServer.renderToString(About())
  response.render('about', {about: aboutHTML})
})

app.all('*', (request, response, next)=> {
  response.status(404).send('Not found...
  ➥ did you mean to go to /about instead?')
})

app.use((error, request, response, next) => {
  console.error(request.url, error)
  response.send('Wonderful, something went wrong...')
})

app.use(errorHandler)

http.createServer(app)
  .listen(3000)

try {
  const options = {
    key: fs.readFileSync('./server.key'),
    cert: fs.readFileSync('./server.crt')
  }
} catch (e) {
  console.warn('Create server.key and server.crt for HTTPS')
}
if (typeof options != 'undefined')
  https.createServer(app, options)
    .listen(443)5
```

Реализует универсальный резервный вариант. Вы не поверите, сколько людей на моих курсах реализуют сервер, переходят по несуществующему URL-адресу и думают, что произошла ошибка, когда в действительности они должны просматривать /about

Загружает ключ и сертификат для SSL/HTTPS[1]

[1] О том, как они генерируются, рассказано в моем сообщении «Easy HTTP/2 Server with Node.js and Express.js», https://webapplog.com/http2-node.

Теперь все должно быть готово к запуску сервера командой `node index.js` или ее сокращенной формой (`node .`), чтобы увидеть ответ сервера при переходе по адресу http://localhost:3000/about.

Если чего-то не хватает или вы получите ошибку при запуске сервера и переходе по адресу, обратитесь к исходному коду проекта в папке ch17/react-express.

> **ПРЕДУПРЕЖДЕНИЕ** Ключ SSL и сертификат необходимы для работы SSL и HTTPS. Код этого примера с GitHib намеренно не включает `server.key` и `server.crt`, потому что конфиденциальная информация (такая, как ключи) не должна закрепляться в системе контроля версий. Вы должны создать собственные ключи, выполнив инструкции по адресу https://webapplog.com/http2-node. Если у вас их нет, код примера только создаст сервер HTTP.

Конечным результатом должна быть правильно сформированная страница HTML с заголовком и телом. В теле должна присутствовать разметка React (например, `data-react-checksum` и `data-reactroot`), как видно из рис. 17.6.

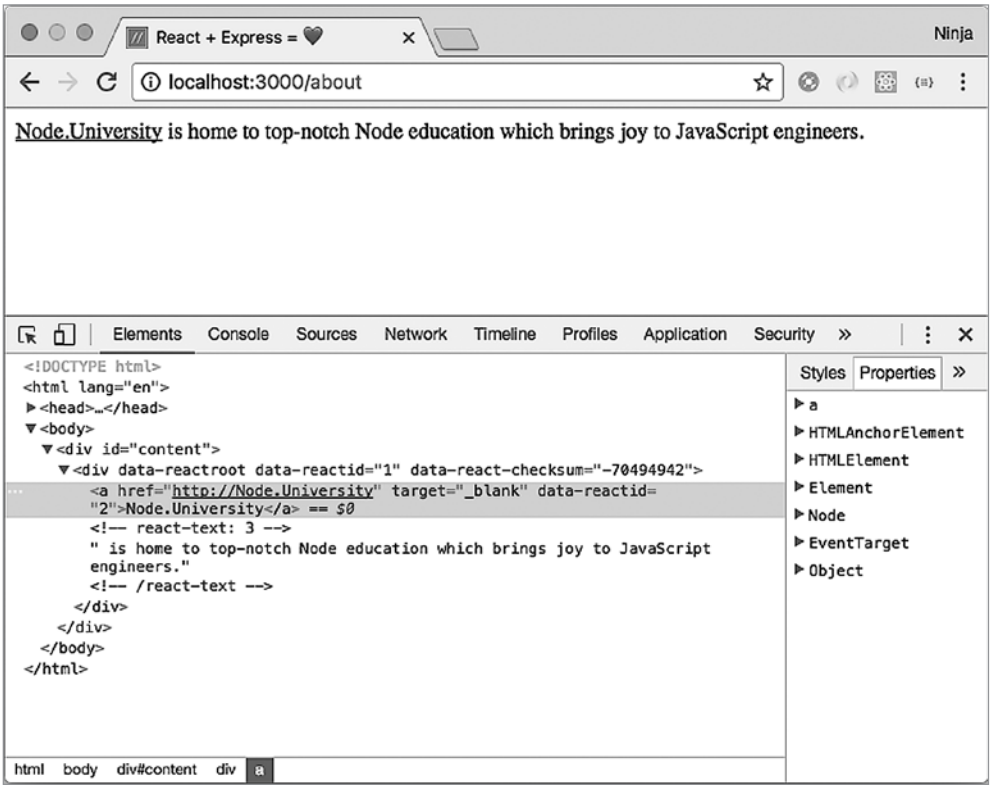


Рис. 17.6. Рендеринг разметки React из структуры Handlebars с использованием Express дает страницу HTML

Почему в этом примере используется рендеринг разметки, а не статические строки HTML и не `express-react-views`? Разметка с контрольными суммами понадобится позднее для React в браузере; это одна из составляющих архитектуры универсального JavaScript.

В следующем разделе мы соберем воедино все, что вы узнали о React в браузере, Express и React в Node для реализации архитектуры универсального JavaScript.

17.4. Универсальный JavaScript с Express и React

В этом разделе объединяется весь материал этой главы (и большей части книги!). Мы выполним рендеринг компонентов на сервере, подключим их к шаблону и обеспечим работу React в браузере.

Для изучения универсального JavaScript мы построим электронную доску сообщений с тремя компонентами: `Header`, `Footer` и `MessageBoard` (рис. 17.7). Компо-

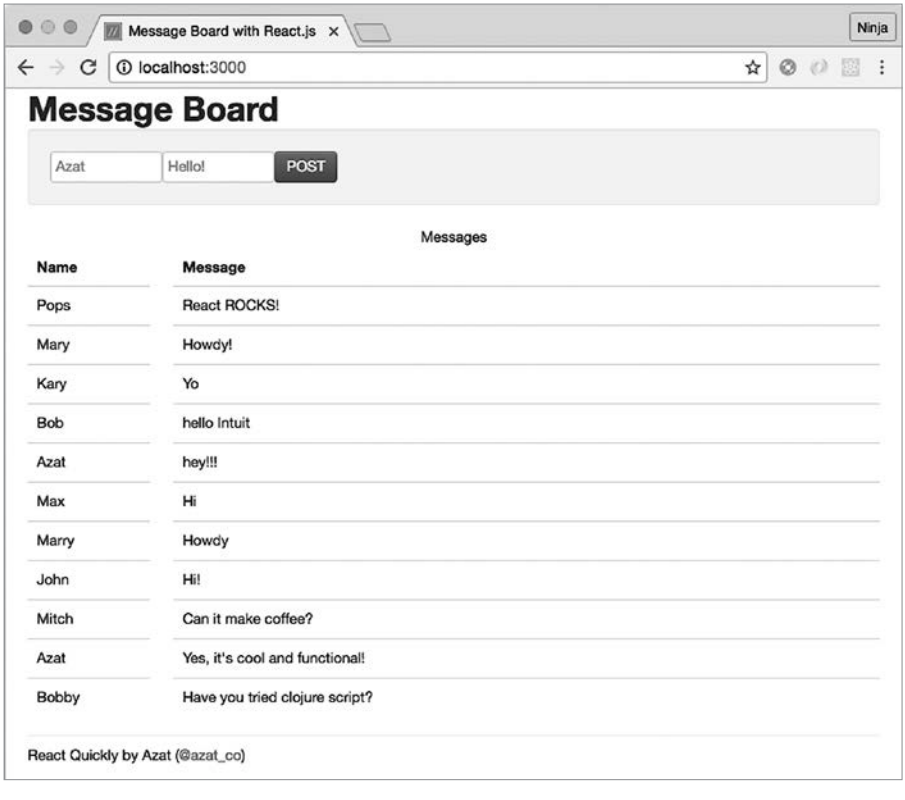


Рис. 17.7. Доска сообщений с формой отправки и списком существующих сообщений

ненты `Header` и `Footer` содержат статическую разметку HTML для вывода текста, а `MessageBoard` содержит форму для публикации сообщений на доске и список сообщений. Приложение использует вызовы AJAX для вывода списка сообщений и передачи новых сообщений серверу, который, в свою очередь, использует базу данных NoSQL MongoDB.

Если говорить точнее, для универсального React следует выполнить следующие действия:

1. Настроить сервер, чтобы он предоставлял данные шаблону и рендерил разметку HTML (компоненты и свойства), например index.js.
2. Создать шаблон, который выводит неэкранированные данные (views/index.hbs).
3. Включить браузерный файл React (`ReactDOM.Render`) в шаблон для взаимодействия с пользователем (client/app.jsx).
4. Создать компоненты `Header`, `Footer` и `MessageBoard`.
5. Подготовить процессы построения с Webpack (webpack.config.js).

В этой схеме задействовано несколько составляющих, взаимодействующих друг с другом: сервер, компоненты, данные и браузер. На рис. 17.8 изображена структура

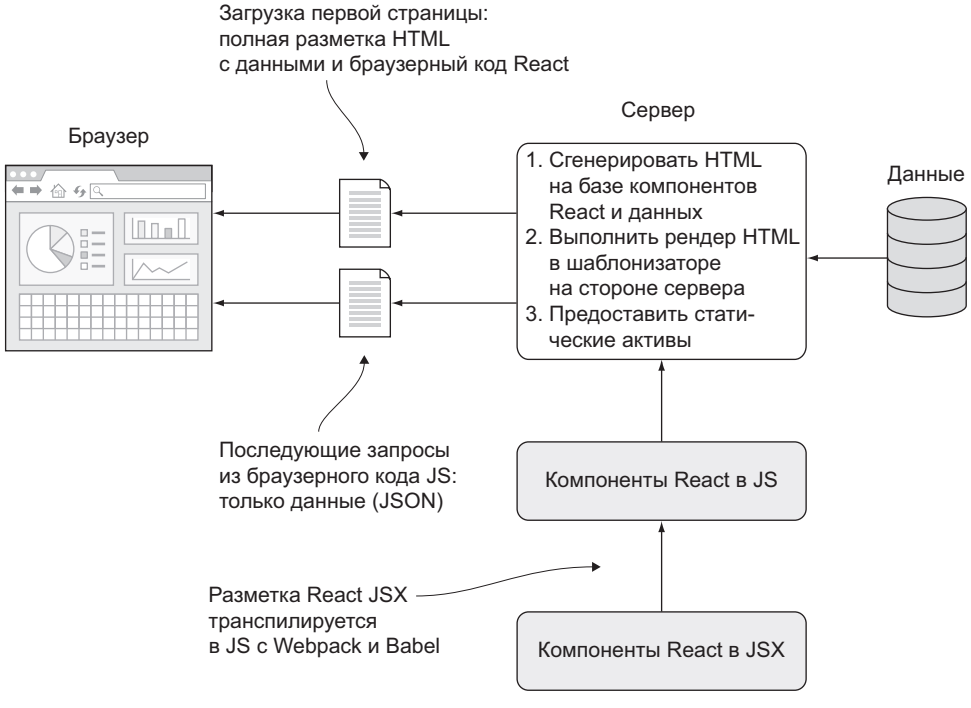


Рис. 17.8. Схема архитектуры универсального JavaScript с React и Express

связей между ними в примере с доской сообщений. Сервер работает как сервер HTTP статических активов и как приложение, которое рендерит разметку HTML на стороне сервера (только для загрузки первой страницы). Браузерный код React обеспечивает взаимодействие с браузерными событиями и последующее сохранение данных (посредством запросов HTTP к серверу) после загрузки исходной страницы.

> **ПРИМЕЧАНИЕ** Для этого примера также необходимо установить и запустить MongoDB. Описание процесса установки можно найти на сайте MongoDB или в приложении Г. После того как установка MongoDB будет завершена, запустите mongod и оставьте процесс работать. Ваш сервер Express сможет связаться с ним по специальному URL-адресу mongodb://local-host:27017/board.

17.4.1. Структура и конфигурация проекта

Проект имеет следующую структуру:

```
/client
  app.jsx   <-- Клиентский/браузерный код
/components
  board.jsx   <-- Код, общий для клиента/браузера и сервера
  footer.jsx
  header.jsx
/node_modules
/public
  /css
  /js   <-- После компиляции и сборки сценариями Webpack
    bundle.js
    bundle.js.map
  /views
    index.hbs
  index.js   <-- Серверный код
  package.json
  webpack.config.js
```

Серверные зависимости включают следующие пакеты (фрагмент файла package.json):

```
...
"dependencies": {
  "babel-register": "6.11.6",   <-- Загружает JSX из Node командой require
  "body-parser": "1.13.2",
  "compression": "1.5.1",
  "errorhandler": "1.4.1",
  "express": "4.13.1",   <-- Использует фреймворк Express
  "hbs": "4.0.0",
  "express-validator": "2.13.0",
  "mongodb": "2.2.6",   <-- Использует MongoDB для хранения сообщений (это драйвер; вам понадобится как драйвер, так и база данных)
  "morgan": "1.6.1",
  "react": "15.5.4",   <-- Использует React для рендера разметки на сервере
  "react-dom": "15.5.4"
},
...
```

Теперь можно заняться настройкой сервера в message-board/index.js.

ПРОМЕЖУТОЧНЫЕ МОДУЛИ EXPRESS

Если вы еще не работали с Express, я хочу сказать несколько слов о промежуточных модулях, использованных в этом проекте. Express не является большим фреймворком, который делает за вас почти все. Напротив, это базовая прослойка, поверх которой программисты Node строят собственные системы — фактически собственные фреймворки. Они точно приспособлены для выполняемой задачи, чего нельзя сказать об универсальных фреймворках. Express и его экосистема плагинов предоставляют лишь самое необходимое. Эти плагины называются «промежуточными модулями», потому что они используют паттерн «промежуточного ПО» (middleware), где Express выполняет функции диспетчера промежуточного кода.

У каждого программиста Express есть любимые промежуточные модули, которыми он пользуется от проекта к проекту. Я обычно начинаю со следующего набора и добавляю новые пакеты в том случае, если возникнет такая необходимость:

- `compression` — автоматически сжимает ответы с применением алгоритма `gzip`. Сжатие сокращает размер ответов и ускоряет загрузку.
- `errorhandler` — простейший обработчик таких ошибок, как 404 и 500.
- `express-validator` — проверяет содержимое входящих запросов. Присутствие этого модуля никогда не повредит.
- `morgan` — ведет журнал запросов на сервере. Поддерживаются разные форматы.
- `body-parser` — обеспечивает автоматический разбор JSON и формата данных `urlencoded` в объектах Node/JS, доступных в `request.body`.

За информацией о сжатии, `body-parser` и `errorhandler`, а также за списком дополнительных промежуточных модулей Express обращайтесь к приложению C, https://github.com/azat-co/cheatsheets/tree/master/express4, или Pro Express.js (http://proexpressjs.com).

17.4.2. Настройка сервера

По аналогии с предыдущими примерами мы сначала реализуем серверную сторону происходящего в index.js, а затем проанализируем пять разделов, чтобы вы видели, как это все работает.

Начнем с полного кода, приведенного в листинге 17.6 (ch17/message-board/index.js).

Листинг 17.6. Серверная часть приложения

```
require('babel-register')({
  presets: [ 'react' ]
})

const express = require('express'),
  mongodb = require('mongodb'),
  app = express(),
  bodyParser = require('body-parser'),
  validator = require('express-validator'),
  logger = require('morgan'),
  errorHandler = require('errorhandler'),
  compression = require('compression'),
  url = 'mongodb://localhost:27017/board',
  ReactDOMServer = require('react-dom/server'),
  React = require('react')

const Header = React.createFactory(require('./components/header.jsx')),
  Footer = React.createFactory(require('./components/footer.jsx')),
  MessageBoard = React.createFactory(require('./components/board.jsx'))

mongodb.MongoClient.connect(url, (err, db) => {
  if (err) {
    console.error(err)
    process.exit(1)
  }

  app.set('view engine', 'hbs')

  app.use(compression())
  app.use(logger('dev'))
  app.use(errorHandler())
  app.use(bodyParser.urlencoded({extended: true}))
  app.use(bodyParser.json())
  app.use(validator())
  app.use(express.static('public'))

  app.set('view engine', 'hbs')

  app.use((req, res, next) => {
    req.messages = db.collection('messages')
    return next()
  })

  app.get('/messages', (req, res, next) => {
    // ...
  })
  app.post('/messages', (req, res, next) => {
    // ...
  })

  app.get('/', (req, res, next) => {
    // ...
  })

  app.listen(3000)
})
```

Определяет отображаемое имя для Imports JSX и компилирует его «на ходу» в JS НОС

Определяет адрес локального экземпляра MongoDB, а также имя базы данных (board)

Подключается к экземпляру MongoDB с использованием URI

Рендерит коллекцию как свойство объекта запроса для удобства обращения в других маршрутах и для их модуляризации

Конфигурация

И снова необходимо использовать `babel-register` для импортирования JSX после установки `babel-register` и `babel-preset-react` из npm:

```
require('babel-register')({
  presets: [ 'react' ]
})
```

В файле index.js реализуется ваш сервер Express. Компоненты импортируются с указанием относительного пути `./components/`:

```
const Header = React.createFactory(require('./components/header.jsx')),
  Footer = React.createFactory(require('./components/footer.jsx')),
  MessageBoard = React.createFactory(require('./components/board.jsx'))
```

Следует знать, что Express.js может использовать практически любой шаблонизатор. Возьмем Handlebars, близкий к обычной разметке HTML. Для включения Handlebars используется следующая команда (предполагается, что `app` — экземпляр Express.js):

```
app.set('view engine', 'hbs')
```

Модуль `hbs` должен быть установлен (у меня он находится в package.json).

Промежуточные модули

Промежуточные модули предоставляют значительную часть функциональности сервера, которую вам пришлось бы реализовать самостоятельно. Ниже перечислены модули, наиболее критичные для этого проекта.

```
// ...
  app.use(compression())
  app.use(logger('dev'))
  app.use(errorHandler())
  app.use(bodyParser.urlencoded({extended: true}))
  app.use(bodyParser.json())
  app.use(validator())
  app.use(express.static('public'))
  // ...
```

Обеспечивает ведение журнала запросов на сервере для упрощения отладки и разработки

Обеспечивает разбор входных данных в формате JSON

Предоставляет доступ ко всем файлам в public (в частности, bundle.js)

Маршруты на стороне сервера

В своем маршруте (допустим, `/`) вы вызываете `render` для views/index.handlebars (`res.render('index')`), потому что по умолчанию для шаблонов используется папка views:

```
app.get('/', (req, res, next) => {
    req.messages.find({}, {sort: {_id: -1}}).toArray((err, docs) => {
      if (err) return next(err)
      res.render('index', data)
    })
  })
```

Вызов `req.message.find()` — метод MongoDB для получения документов. Хотя для того, чтобы этот пример работал без каких-либо изменений, на машине должен быть установлен и запущен экземпляр MongoDB, я не хочу навязывать вам свои предпочтения по части баз данных. Вызовы MongoDB легко заменяются любыми другими по вашему выбору. У большинства современных реляционных СУБД и баз данных NoSQL существуют драйверы Node; у большинства из них даже имеются библиотеки ORM/ODM, написанные на Node. Следовательно, вы можете спокойно проигнорировать мой вызов базы данных, если вы не планируете использовать MongoDB. А если вы все же хотите использовать MongoDB, в приложении Г приведена краткая сводка. Суть заключается в том, что в обработчике запроса вы обращаетесь с вызовом к внешнему сервису (например, используете `axios` для получения информации о пользователях Facebook) или используете базу данных (MongoDB, PostgreSQL и т. д.). Конкретный способ получения данных в Node не является частью этой главы.

Здесь самое важное в отношении универсального React — метод `res.render()` (ch17/message-board/index.js), приведенный в листинге 17.7. Этот метод `render()` представляет собой специальную функцию Express для шаблонов. Он получает два аргумента. В первом передается имя шаблона: index.hbs из каталога views. Во втором аргументе `res.render()` передаются данные, которые будут использоваться в шаблонах. Все данные передаются шаблону ch17/message-board/view/index.hbs (расширение .hbs не является обязательным).

Листинг 17.7. Рендеринг разметки HTML, сгенерированной из компонентов React

```
...
  app.set('view engine', 'hbs')
  ...

  app.get('/', (req, res, next) => {
    req.messages.find({},
      {sort: {_id: -1}}).toArray((err, docs) => {
      if (err) return next(err)
      res.render('index', {
        header: ReactDOMServer.renderToString(Header()),
        footer: ReactDOMServer.renderToString(Footer()),
        messageBoard: ReactDOMServer.renderToString(MessageBoard({
          messages: docs
        })),
        props: '<script type="text/javascript">var messages='
          +JSON.stringify(docs)
          +'</script>'
      })
    })
  })
```

Применяет шаблонизатор Handlebars

Запрашивает у MongoDB массив сообщений, отсортированных в обратном порядке по ObjectID

Отправляет строку HTML, сгенерированную из Header

Отправляет список сообщений браузерному коду React

Отправляет строку HTML, сгенерированную из MessageBoard, со списком сообщений (docs) в свойстве

Отправляет строку HTML, сгенерированную из Footer

К этому моменту у вас имеется сервер Express, который рендерит шаблон Handlebars с тремя строками HTML из компонентов React. Само по себе это не впечатляет; то же самое можно было сделать и без React. Вы могли воспользоваться Handlebars, Pug, Mustache или любым другим шаблонизатором для рендеринга всего, не только структуры. Зачем здесь нужен React? Вы будете использовать React в браузере, а браузерный код React возьмет серверную разметку HTML и добавит все события и состояния — все самое интересное. Вот зачем!

Работа над сервером еще не закончена. Для этого примера необходимо реализовать два API:

- `GET /messages` — получение списка сообщений из базы данных.
- `POST /messages` — создание нового сообщения в базе данных.

Эти маршруты будут использоваться браузерным кодом React при выдаче запросов AJAX/XHR для получения/отправки данных. Код маршрутов размещается в Express в файле index.js:

```
app.get('/messages', (req, res, next) => {
    req.messages.find({},
      {sort: {_id: -1}}).toArray((err, docs) => {
      if (err) return next(err)
      return res.json(docs)
    })
  })
```

Маршрут для обработки создания сообщений (`POST /messages`) будет использовать `express-validator` для проверки наличия входных данных (`notEmpty()`). `express-validator` — удобный промежуточный модуль, позволяющий определить самые разнообразные правила проверки.

ПРЕДУПРЕЖДЕНИЕ Проверка ввода жизненно важна для защиты ваших приложений. Разработчики работают с кодом и с системой: они пишут код, они знают, как он работает и какие данные он поддерживает. По этой причине они бессознательно склонны необъективно относиться к данным, которые поставляются их приложениям, что может создать уязвимости. Всегда проверяйте данные на стороне сервера. Рассматривайте каждого пользователя как злоумышленника или невнимательного оболтуса, который никогда не читает ваши инструкции и всегда отправляет некорректные данные.

Маршрут также использует ссылку на базу данных из `req.messages` для вставки нового сообщения:

```
app.post('/messages', (req, res, next) => {
    req.checkBody('message',
      'Invalid message in body').notEmpty()
    req.checkBody('name', 'Invalid name in body').notEmpty()
    var errors = req.validationErrors()
    if (errors) return next(errors)
    req.messages.insert(req.body, (err, result) => {
```

Проверяет, что в теле запроса присутствует сообщение

Вставляет тело запроса в базу данных

```
    if (err) return next(err)
    return res.json(result.ops[0])
  })
})
```

Выводит идентификатор нового документа, автоматически сгенерированный базой данных

NODE-DEV

Как упоминалось ранее, я рекомендую использовать nodemon или другую аналогичную программу, например node-dev. node-dev отслеживает изменения в файлах и перезапускает сервер при обнаружении изменений. Программа может сэкономить вам много часов работы! Чтобы установить node-dev, выполните следующую команду:

```
npm i node-dev@3.1.3 --save-dev
```

Добавьте в package.json команду node-dev . для запуска сценария npm:

```
...
  "scripts": {
    ...
    "start": "./node_modules/.bin/webpack && node-dev ."
  },
  ...
```

Команда запуска кажется примитивной по сравнению с предыдущим разделом, когда мы использовали протокол HTTPS:

```
app.listen(3000)
```

Разумеется, в него можно добавить HTTPS и изменить номер порта или же взять номер порта из переменных среды.

Помните, что корневой маршрут / обрабатывает все запросы GET к / или к http://localhost:3000/ в данном случае. Реализация маршрута приведена в листинге 17.7 (ch17/message-board/view/index.hbs). Маршрут использует шаблон с именем index в res.render(). А теперь давайте реализуем этот шаблон.

17.4.3. Структурные шаблоны на стороне сервера с использованием Handlebars

Для генерирования React HTML на сервере может использоваться любой шаблонизатор. Handlebars — хороший вариант, потому что он достаточно близок к HTML; это означает, что переход с HTML на этот шаблонизатор потребует небольших изменений. Ниже приведен файл index.hbs для Handlebars:

```
<!DOCTYPE html>
<html lang="en">
  <head>
```

```
    <!-- теги метаданных и CSS -->
  </head>

  <body>
  <div class="container-fluid">
    <!-- header -->
    <!-- props -->
    <!-- messageBoard -->
    <!-- footer -->
  </div>
  <script type="text/javascript" src="/js/bundle.js"></script>
  </body>
</html>
```

Вы можете использовать тройные фигурные скобки (`{{{...}}}`) для вывода компонентов и свойств (неэкранированный вывод), например HTML. Так, `{{{props}}}` будет выводить тег `<script/>`, чтобы вы могли определить в нем переменную `messages`. Код index.hbs для генерирования неэкранированной строки HTML для `props` выглядит так:

```
<div>{{{props}}}</div>
```

Остальные данные выводятся аналогичным образом:

```
<div id="header">{{{header}}}</div>
...
<div>{{{props}}}</div>
...
<div class="row-fluid" id="message-board" />{{{messageBoard}}}</div>
...
<div id="footer">{{{footer}}}</div>
```

А вот как выводится строка HTML из компонента `Header` в Handlebars (ch17/message-board/views/index.hbs).

Листинг 17.8. Вывод разметки HTML, сгенерированной React в Handlebars

```
...
  <div class="container-fluid">
    <div class="row-fluid">
      <div class="span12">
        <div id="header">{{{header}}}</div>
      </div>
    </div>
    ...
```

Как насчет данных? Чтобы пользоваться преимуществами комбинации React на стороне сервера с браузерным кодом React, при создании элементов React в браузере и на сервере должны использоваться одни и те же данные. Вы можете передать данные от React на сервере к React в браузере без необходимости вызывать AJAX, для чего данные встраиваются в переменную JS прямо в HTML!

При передаче `header`, `footer` и `messageBoard` вы можете добавить `props` в маршрут Express `/`. В index.hbs выведите значения в тройных фигурных скобках и включите сценарий js/bundle.js, который будет сгенерирован Webpack позднее (ch17/message-board/views/index.hbs).

Листинг 17.9. Серверная структура для рендера HTML из компонентов React

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <title>Message Board with React.js</title>
    <meta name="description" content="Message Board" />
    <meta name="author" content="Azat Mardan" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <link type="text/css" rel="stylesheet" href="/css/bootstrap.min.css" />
    <link type="text/css" rel="stylesheet"
    ➥ href="/css/bootstrap-responsive.min.css" />
  </head>
  <body>
    <div class="container-fluid">
      <div class="row-fluid">
        <div class="span12">
          <div id="header">{{{header}}}</div>
        </div>
      </div>
      <div>{{{props}}}</div>
      <div class="row-fluid">
        <div class="span12">
          <div id="content">
            <div class="row-fluid" id="message-board" />{{{messageBoard}}}</div>
        </div>
      </div>
      <div class="row-fluid">
        <div class="span12">
          <div id="footer">{{{footer}}}</div>
        </div>
      </div>
    </div>
    <script type="text/javascript" src="/js/bundle.js"></script>
  </body>
</html>
```

Выводит разметку HTML, сгенерированную из компонента Header

Выводит разметку HTML, содержащую элемент <script> со списком сообщений в виде массива

Включает браузерный код React

Шаблон включает стилевое оформление Twitter Bootstrap, но для проекта примера универсального JavaScript это не так важно. В ваших шаблонах используются некоторые переменные (`header`, `messageBoard`, `props` и `footer`), которые должны предоставляться в вызове `render()` обработчика запроса Express. Для напоминания приведу код index.js, реализованный ранее (листинг 17.7, ch17/message-board/view/index.hbs), который использует предыдущий шаблон под именем `index` (сокращение для index.hbs):

```
res.render('index', {
  header: ReactDOMServer.renderToString(Header()),
  footer: ReactDOMServer.renderToString(Footer()),
  messageBoard:
    ReactDOMServer.renderToString(MessageBoard({messages: docs})),
  props: '<script type="text/javascript">var messages='+JSON.stringify(docs)+
  ➥ '</script>'
})
```

Значения будут генерироваться из компонентов React. Таким образом вы будете использовать одни и те же компоненты на сервере и в браузере. Возможность простого рендеринга на сервере (с Node) — одна из сильнейших сторон React.

А теперь перейдем к переменным `props`, `header`, `footer` и т. д.

17.4.4. Формирование компонентов React на сервере

Наконец-то мы переходим к тому, чем занимались во всех предыдущих главах, — созданию компонентов React. Согласитесь, приятно вернуться к знакомому занятию. Но откуда берутся эти компоненты? Они размещаются в папке components. Как я упоминал ранее, эти компоненты будут использоваться в браузере и на сервере; вот почему они находятся в отдельной папке, а не создаются в клиенте. (Также папке компонентов часто присваиваются имена shared и common.)

Чтобы эти компоненты были доступны, каждый из них должен содержать переменную `module.exports`, которой присваивается значение — класс компонента или функция без состояния. Например, вы включаете React, реализуете класс или функцию, а затем экспортируете `Header` следующим образом:

```
                                    Хотя React в коде не используется
                                    напрямую, это необходимо для JSX
const React = require('react')
const Header = () => {   ←— Объявляет компонент без состояния
  return (
    <h1>Message Board</h1>
  )
}

module.exports = Header   ←— Экспортирует компонент без состояния
```

Доска сообщений будет использовать вызовы AJAX/XHR для получения списка сообщений и публикации нового сообщения. Вызовы находятся в board.jsx. Файл включает компонент `MessageBoard`. Данный компонент является контейнерным, так что вызовы находятся в этом компоненте.

Интересно, что в `MessageBoard` вызовы AJAX совершаются в `componentDidMount()` — ведь этот метод жизненного цикла никогда не будет вызываться на сервере (ch17/message-board/components/board.jsx)!

Листинг 17.10. Получение и отправка сообщений

```
const request = require('axios')
const url = 'http://localhost:3000/messages'
const fD = ReactDOM.findDOMNode
...
class MessageBoard extends React.Component {
  constructor(ops) {
    super(ops)
    this.addMessage = this.addMessage.bind(this)
    if (this.props.messages)
      this.state = {messages: this.props.messages}
  }
  componentDidMount() {
    request.get(url, (result) => {
      if(!result || !result.length){
        return;
      }
      this.setState({messages: result})
    })
  }
  addMessage(message) {
    let messages = this.state.messages
    request.post(url, message)
      .then(result => result.data)
      .then((data) =>{
        if(!data){
          return console.error('Failed to save')
        }
        console.log('Saved!')
        messages.unshift(data)
        this.setState({messages: messages})
    })
  }
  render() {
    return (
      <div>
        <NewMessage messages={this.state.messages} addMessageCb=
          {this.addMessage} />
        <MessageList messages={this.state.messages} />
      </div>
    )
  }
}
```

Создает переменную для адреса сервера. Значение может быть изменено позднее

Выдает запрос GET из axios и в случае успеха обновляет состояние списком сообщений

Выдает запрос POST из axios и в случае успеха добавляет сообщение в список с обновлением состояния

Передает метод добавления сообщений в презентационный компонент NewMessage, который создает форму и слушателей сообщений

Реализация `NewMessage` и `MessageList` содержится в том же файле (ch17/message-board/components/board.jsx); я не буду утомлять вас описаниями. Это презентационные компоненты, содержащие минимум логики — только описание пользовательского интерфейса в форме JSX.

Мы разобрались с рендером разметки React (и структуры) HTML на сервере. Теперь займемся синхронизацией разметки с React в браузере; в противном случае

сообщения добавляться не будут — ведь не будет интерактивных браузерных событий JavaScript!

17.4.5. Клиентский код React

Если остановить реализацию в этой точке, у вас будет только статическая разметка в результате рендера компонентов React на сервере. Новые сообщения сохраняться не будут, потому что событие `onClick` для кнопки POST работать не будет. Необходимо подключить React в браузере и продолжить с того места, на котором остановился рендеринг статической разметки на сервере.

Файл app.jsx создается только для браузера. Он не будет выполняться на сервере (в отличие от компонентов). Здесь и следует разместить вызовы `ReactDOM.render()` для обеспечения работы React в браузере:

```
ReactDOM.render(<MessageBoard messages={messages}/>,
  document.getElementById('message-board')
)
```

Вам также потребуется использовать глобальную переменную `messages` как свойство `MessageBoard`. Значение свойства `messages` будет заполняться шаблоном на стороне сервера и данными `{{{props}}}` (см. раздел 17.4.3). Другими словами, массив сообщений `messages` будет заполняться из index.hbs при получении шаблоном данных из переменной `props` в маршруте Express.js `/`.

Несовпадение значений свойства `messages` компонента `MessageBoard` на сервере и в браузере приведет к тому, что в браузере React перерисует весь компонент, потому что React в браузере будет считать представления различающимися. Во внутренней реализации React использует атрибут `checksum` для сравнения данных, уже находящихся в DOM (в результате рендеринга на стороне сервера), с теми данными, которые будут у React в браузере. React использует значение `checksum`, потому что такая проверка выполняется быстрее сравнения деревьев (что заняло бы значительное время).

В файле app.js необходимо включить некоторые библиотеки клиентской части, а затем выполнить рендер компонентов в DOM (ch17/message-board/client/app.jsx).

Листинг 17.11. Рендеринг клиентских компонентов React в браузере

```
const React = require('react')
const ReactDOM = require('react-dom')

const Header = require('../components/header.jsx')
const Footer = require('../components/footer.jsx')
const MessageBoard = require('../components/board.jsx')

ReactDOM.render(<Header />, document.getElementById('header'))
ReactDOM.render(<Footer />, document.getElementById('footer'))
ReactDOM.render(<MessageBoard messages={messages}/>,
➥ document.getElementById('message-board'))
```

Браузерный код получается совсем крошечным!

17.4.6. Настройка Webpack

Остается сделать последний шаг: настроить Webpack для сборки браузерного кода в один файл, управления зависимостями и преобразования кода JSX. Сначала необходимо настроить Webpack так, как описано ниже: с точкой входа client/app.jsx, выводом в public/js в каталоге проекта и с использованием загрузчиков Babel. Параметр `devtool` обеспечивает вывод правильной нумерации строк исходного кода в Chrome DevTools (не строк из откомпилированного кода JS):

```
module.exports = {
  entry: './client/app.jsx',
  output: {
    path: __dirname + '/public/js/',
    filename: 'bundle.js'
  },
  devtool: '#sourcemap',
  stats: {
    colors: true,
    reasons: true
  },
  module: {
    loaders: [
      {
        test: /\.jsx?$/,
        exclude: /(node_modules)/,
        loader: 'babel-loader'
      }
    ]
  }
}
```

Чтобы преобразовать JSX в JS, можно воспользоваться `babel-preset-react` и указать конфигурацию Babel в package.json:

```
...
  "babel": {
    "presets": [
      "react"
    ]
  },
  ...
```

Зависимости на стороне клиента (для React в браузере), такие как Babel и Webpack в package.json, остаются зависимостями разработки, потому что Webpack упакует все необходимое в bundle.js. Следовательно, на стадии выполнения они не понадобятся:

```
{
  ...
  "devDependencies": {
    "axios": "0.13.1",
    "babel-core": "6.10.4",
```

```
    "babel-jest": "13.2.2",
    "babel-loader": "6.2.4",
    "babel-preset-react": "6.5.0",
    "node-dev": "3.1.3",
    "webpack": "1.13.1"
  }
}
```

СОВЕТ Проследите за тем, чтобы в вашем проекте использовались именно те версии, которые приведены здесь. Иначе все обновления, которые выйдут за то время, пока я пишу этот абзац, нарушат работоспособность вашего проекта — и это только наполовину шутка!

Кроме того, пока вы работаете с package.json, добавьте сценарий `npm build` (это не обязательно, но удобно):

```
  ...
  "scripts": {
    ...
    "build": "./node_modules/.bin/webpack"
  },
...
```

Лично мне нравится режим отслеживания изменений для Webpack (`-w`). В `package.json` можно добавить ключ `-w` в сценарий `npm build`:

```
...
"scripts": {
  "build": "./node_modules/.bin/webpack -w",
  ...
},
...
```

Соответственно, при каждом выполнении команды `npm run build` Webpack использует Babel для преобразования JSX в JS и объединяет все файлы с их зависимостями в один огромный шар. В данном случае он будет помещен в /public/js/app.js.

Благодаря включению в шаблоне views/index.hbs, непосредственно перед завершающим тегом `</body>`, браузерный код будет работать (следующая строка взята из шаблона):

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

Запуская эту задачу по умолчанию командой `npm run build`, я получаю следующий вывод:

```
Hash: 1d4cfcb6db55f1438550
Version: webpack 1.13.1
Time: 733ms
        Asset    Size  Chunks             Chunk Names
    bundle.js  782 kB       0  [emitted]  main
bundle.js.map  918 kB       0  [emitted]  main
    + 200 hidden modules
```

Если вы получите другое сообщение или уведомление об ошибке, сравните свой проект с кодом на странице www.manning.com/books/react-quickly или https://github.com/azat-co/react-quickly/tree/master/ch17.

17.4.7. Запуск приложения

Собственно, это все, что необходимо сказать о рендере компонентов React.js в приложениях Express.js. В типичной ситуации вам понадобится только следующее (предполагается, что процесс сборки и компоненты уже готовы):

- Шаблон для вывода неэкранированных данных.
- Вызов `res.render()` для заполнения шаблона данными и его рендера (компоненты, свойства и т. д.).
- Включение браузерного файла React (с `ReactDOM.Render`) в шаблон для обеспечения интерактивности.

Все еще неуверенно чувствуете себя с универсальным Express и React? В таком случае загрузите протестированный, заведомо работающий код проекта на сайте www.manning.com/books/react-quickly или https://github.com/azat-co/react-quickly/tree/master/ch17/message-board и поэкспериментируйте. Например, удалите код из app.js, чтобы заблокировать React в браузере (и отключить всю интерактивность — например щелчки), или удалите код в index.js для блокировки React на сервере (небольшая задержка при загрузке страницы).

Чтобы запустить проект, убедитесь в том, что в системе работает MongoDB (`$ mongod`; за дополнительными инструкциями обращайтесь к приложению Г). В папке проекта выполните следующие команды:

```
$ npm install
$ npm start
```

Не забывайте либо настраивать Webpack для проведения сборки в режиме отслеживания изменений (`npm run build`), либо перезапускать приложение при каждом внесении изменения в браузерный код.

Откройте в браузере адрес http://localhost:3000. Вы увидите доску сообщений (рис. 17.9). Если проанализировать процесс загрузки страницы (Chrome DevTools), вы увидите, что первая загрузка проходит быстрее, потому что разметка HTML рендерится на сервере.

Попробуйте закомментировать в ch17/message-board/index.js код, отвечающий за рендеринг разметки на стороне сервера, и сравнить данные хронометража на вкладке Network. Обратите внимание на ресурс `localhost` (первая загрузка страницы и рендеринг на стороне сервере) и вызов XHR GET к `/messages`. Мои результаты для `localhost` были намного быстрее, как видно из рис. 17.10.

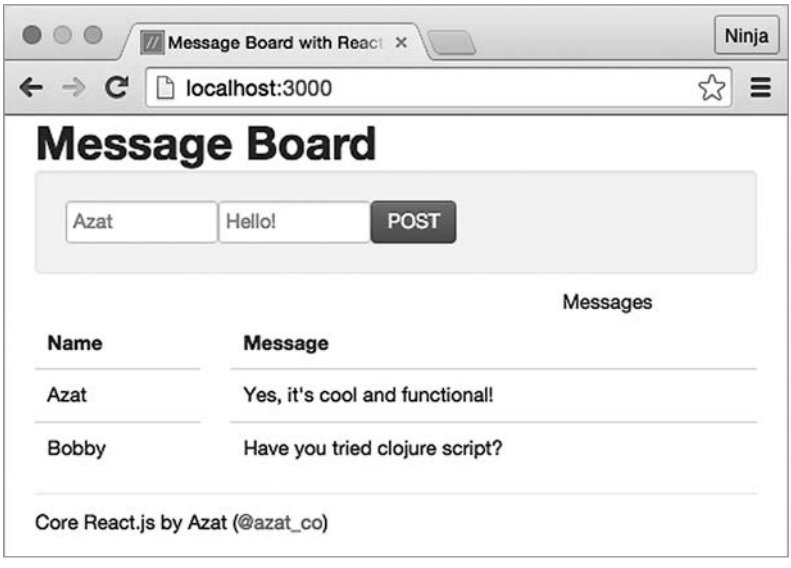


Рис. 17.9. Универсальное приложение с рендером HTML на сервере и в браузере

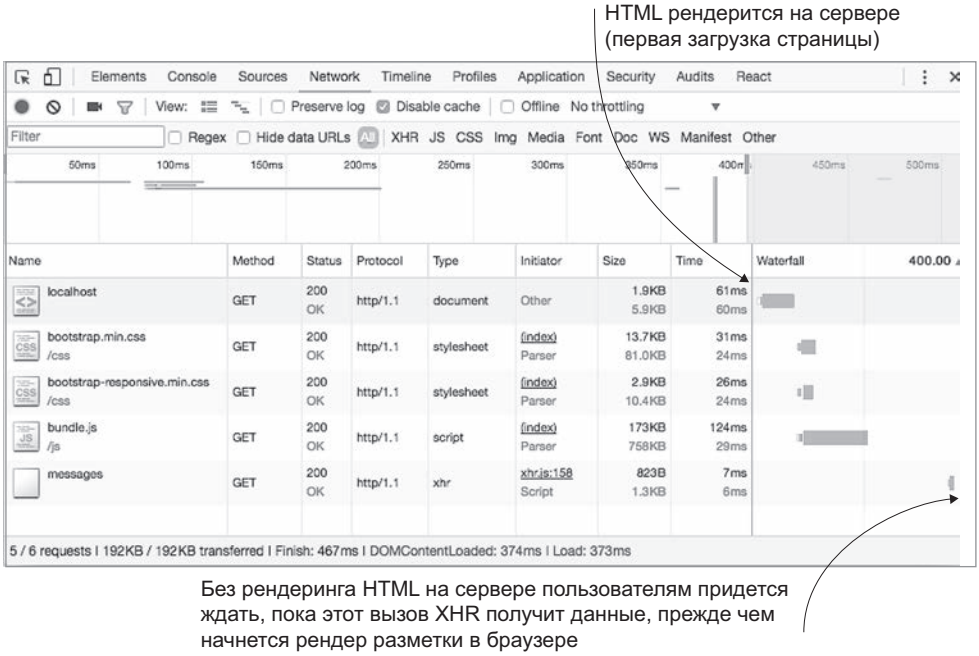


Рис. 17.10. Загрузка серверной разметки HTML происходит в 10 раз быстрее, чем полная загрузка, которая замедляется из-за bundle.js

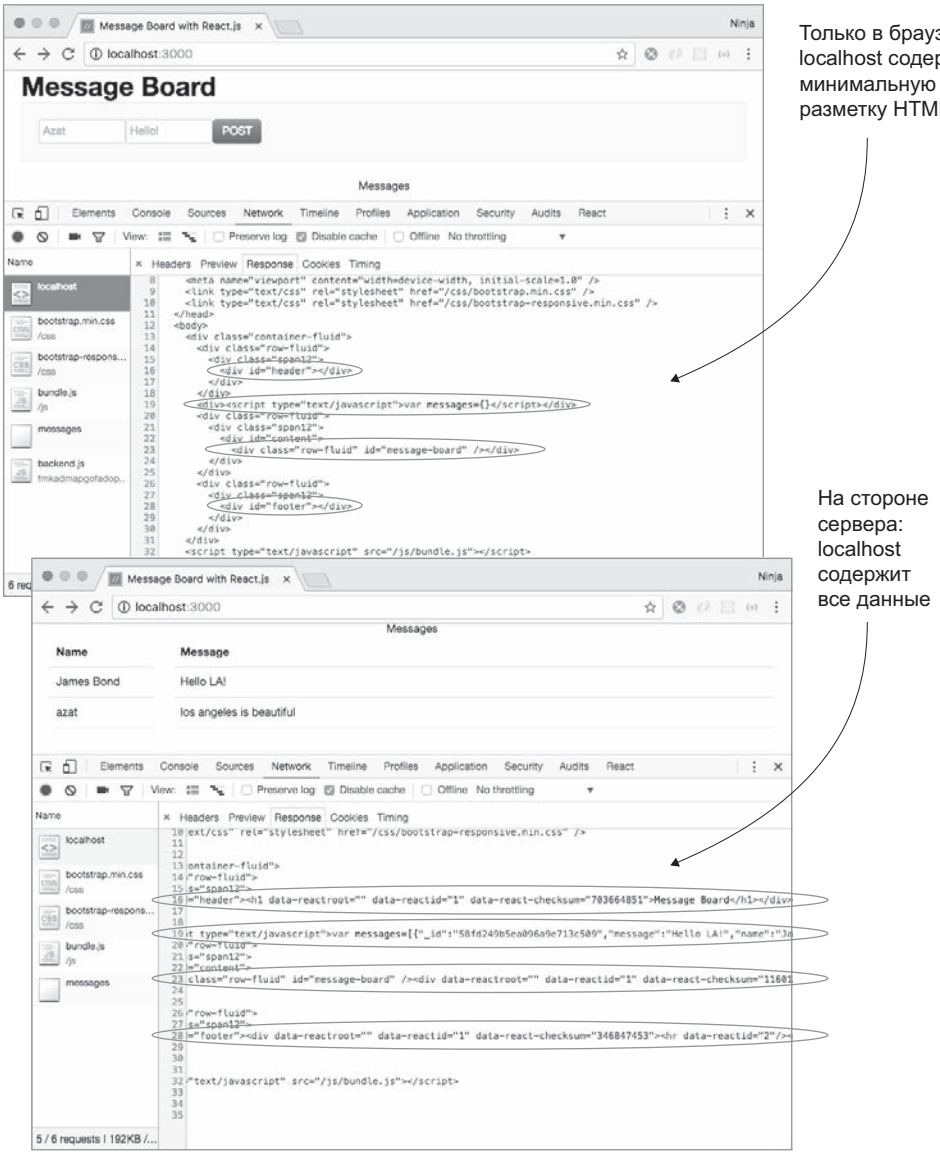


Рис. 17.11. Localhost (первый ответ) для рендеринга разметки только в браузере (наверху) и рендера разметки на стороне сервера (внизу)

Конечно, большую часть времени загрузки занимает bundle.js. В конце концов, файл содержит более 200 модулей! Запрос `GET /messages` много времени не занимает — всего лишь несколько миллисекунд. При этом при вызове `localhost` видят всё на странице. И наоборот, без изоморфного/универсального кода пользователи увидят полностью

сформированную разметку HTML только после сообщений `GET /messages`, а также после того, как React в браузере отрендерит HTML на стороне клиента.

Проанализируем приложение с другой точки зрения: сравним универсальный рендер разметки с браузерным, расположив их рядом друг с другом. На рис. 17.11 показаны результаты для `localhost`. С универсальным подходом `localhost` располагает всеми данными, и загрузка занимает всего 20–30 мс. При использовании React только в браузере `localhost` располагает только минимальной заготовкой HTML, поэтому пользователям приходится ждать приблизительно в 10 раз дольше. Любая задержка, превышающая 150 мс, обычно заметна для пользователей.

Вы можете поэкспериментировать, закрывая комментарием команды рендеринга в index.js (Express.js) или app.jsx (React в браузере). Например, если закомментировать `Header` на стороне сервера, но оставить рендер `Header` в браузере, возможно, пройдет незначительное время перед появлением `Header`.

Кроме того, если закомментировать передачу переменной `props` на сервере или изменить ее значение, React в браузере обновит DOM после получения списка сообщений для `axios`. React выдаст предупреждение о несовпадении контрольных сумм.

УНИВЕРСАЛЬНАЯ МАРШРУТИЗАЦИЯ И ДАННЫЕ

Рано или поздно ваше приложение начнет расти, и вам придется использовать такие библиотеки, как React Router и Redux, для маршрутизации данных (см. главы 13 и 14). Интересно, что эти библиотеки уже поддерживают Node, а React Router даже поддерживает Express. Например, маршруты React Router можно передавать Express для поддержки на стороне сервера (match и RouterContext) для рендера разметки на стороне сервера:

```
const { renderToString}=require('react-dom/server')
const { match, RouterContext } = require ('react-router')
const routes = require('./routes')
// ...
app.get('/', (req, res) => {
  match({ routes,
    location: req.url
  },
    (error,
    redirectLocation,
    renderProps) => {
    // ...
    res
      .status(200)
      .send(renderToString(
        <RouterContext {...renderProps} />
      ))
  })
}
```

Использует специальный метод из React Router

Передает location/URL методу React Router

Рендерит строку HTML с использованием специального компонента React Router и свойств

Redux содержит метод `createStore()` (глава 14), который может использоваться на стороне сервера в промежуточных модулях Express для предоставления хранилища данных. Например, для компонента App код на стороне сервера с Redux будет выглядеть примерно так:

```
const { createStore } = require('redux')
const { Provider } = require('react-redux')
const reducers = require('./modules')
const routes = require('./routes')

// ...
app.use((req, res) => {
  const store = createStore(reducers)
  const html = renderToString(
    <Provider store={store}>
      <App/>
    </Provider>
  )

  const preloadedState = store.getState()

  res.render('index', {html, preloadedState})
})
```

Создает новый экземпляр хранилища Redux

Рендерит строку из компонента

Активизирует хранилище

Обращается к исходному состоянию из хранилища Redux

Рендерит страницу в клиенте с использованием HTML и данных

Шаблон `index` выглядит так:

```
<div id="root">${html}</div>
<script>
  window.__PRELOADED_STATE__ = ${JSON.stringify(preloadedState)}
</script>
<script src="/static/bundle.js"></script>
```

Redux использует тот же подход, который вы использовали для доски сообщений: рендеринг HTML и данных в теге `<script>`.

Полный пример с объяснениями доступен по адресу http://mng.bz/F5pb и http://mng.bz/Edyx.

На этом завершается рассмотрение изоморфного, или универсального, JavaScript. Единство кода и повторное использование кода — огромные преимущества, которые повышают эффективность труда разработчиков и позволяют получать больше удовольствия от работы!

17.5. Вопросы

1. Какой метод используется для рендера компонента React на сервере?
2. Рендеринг первой страницы на сервере повышает быстродействие. Да или нет?

3. Синтаксис модулей CommonJS и Node.js, использующий `require()` (с Webpack), позволяет импортировать модули npm в браузерном коде. Да или нет?
4. Какая из следующих конструкций используется для вывода неэкранированных строк в Handlebars: `<%...%>`, `{{...}}`, `{{{...}}}` или `dangerouslySetInnerHTML=...`?
5. Где лучше всего размещать вызовы AJAX/XHR в браузерном коде React, чтобы они не срабатывали на сервере?

17.6. Итоги

- Чтобы использовать React и рендерить разметку на сервере, вам понадобится модуль `react-dom/server` и метод `renderToString()`.
- Для синхронизации серверной разметки HTML React с браузером данные должны совпадать. React использует контрольные суммы для сравнения.
- Различия между `renderToString()` и `renderToStaticMarkup()` заключаются в том, что `renderToString()` включает контрольные суммы, позволяющие React в браузере заново использовать HTML, а `renderToStaticMarkup()` — нет.
- В архитектуре универсального JS вы выполняете рендер React на сервере, передаете React в браузере те же данные и выполняете рендер компонентов React в браузере.
- Тройные фигурные скобки `{{{html}}}` используются для вывода неэкранированного контента HTML в Handlebars.

17.7. Ответы

1. `ReactDOMServer.renderToString()`. `renderToStaticMarkup()` не генерирует контрольные суммы.
2. Да. Вы получите все данные при первой загрузке страницы, не дожидаясь `bundle.js` и запросов AJAX.
3. Да. С Webpack `require()` и синтаксис `module.exports` могут использоваться без дополнительной настройки. Просто установив точку входа в `webpack.config.js`, вы заставите Webpack обойти все зависимости и включить только те, которые действительно необходимы.
4. `{{{...}}}` — правильный синтаксис. Для экранированных переменных используйте `{{data}}`, чтобы гарантировать безопасное использование.
5. `componentDidMount()`, потому что этот метод никогда не вызывается при рендеринге разметки на сервере.

18 Проект: создание книжного магазина с React Router

Посмотрите вступительный видеоролик к этой главе, отсканировав QR-код или перейдя по ссылке http://reactquickly.co/videos/ch18.

ЭТА ГЛАВА ОХВАТЫВАЕТ СЛЕДУЮЩИЕ ТЕМЫ:

- Структура проекта и конфигурация Webpack.
- Основной файл HTML.
- Создание компонентов.
- Запуск проекта.

Проект в этой главе прежде всего демонстрирует использование React Router, некоторых возможностей ES6 и Webpack. В этом проекте мы построим простой интернет-магазин для продажи книг (рис. 18.1).

Вы узнаете, как создать маршрутизацию в браузере, а также освоите некоторые приемы работы с React Router, в частности:

- как передать данные маршруту и как работать с ними;
- как обращаться к параметрам URL;
- как создавать модальные окна с изменяющимися URL;
- как использовать макеты с вложенными маршрутами.

Проект включает несколько экранов с разными маршрутами:

- Домашняя страница (`/`) — магазин со списком книг.
- Страница продукта (`/product/:id`) — страница отдельного продукта.

- Корзина (`/cart`) — веб-страница с названиями и количеством книг, выбранных пользователем.
- Оформление заказа (`/checkout`) — готовый к печати счет со списком книг.

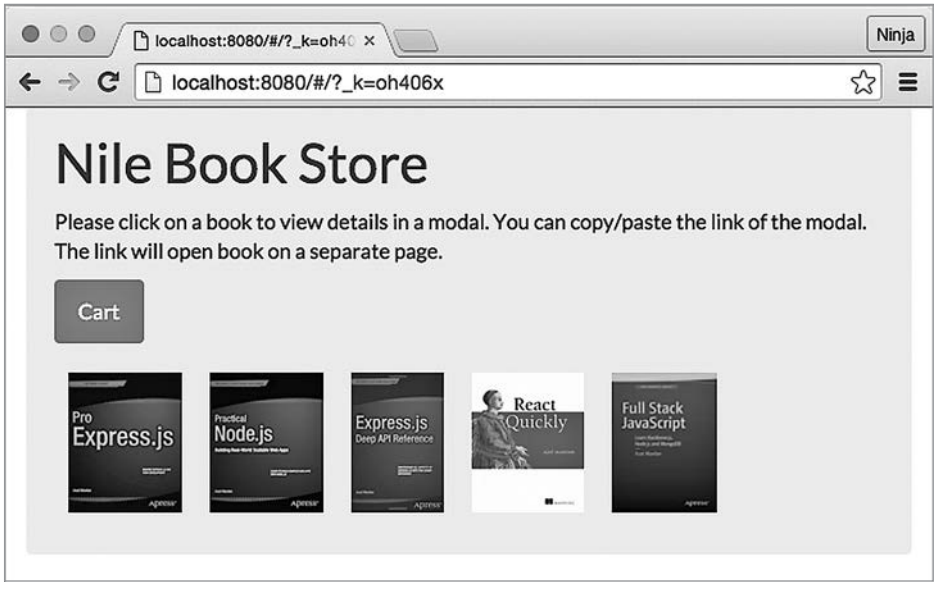


Рис. 18.1. Домашняя страница книжного магазина со списком книг

Информация о продукте берется из массива данных, хранящихся в одном из файлов (ch18/nile/jsx/app.js; см. структуру проекта в следующем разделе). Страница продукта может отображаться в модальном диалоговом окне или на отдельной странице. Если щелкнуть на изображении продукта на домашней странице, откроется модальное диалоговое окно; например, на рис. 18.2 изображено модальное диалоговое окно с подробной информацией об этой книге.

URL-адрес состоит из пути `/products/3` и идентификатора после знака `#`, используемого для отслеживания состояния. Ссылку можно передать другим пользователям; если открыть ее в новом окне/вкладке, это будет нормальный экран, а не модальное диалоговое окно (рис. 18.3). Модальные окна полезны при перемещении по списку, если вы не хотите потерять контекст при переходе к новой странице. Но когда вы передаете прямую ссылку на продукт, никакого контекста или списка нет — все внимание пользователя должно быть обращено на продукт.

Схема реализации интерфейса книжного магазина состоит из следующих этапов:

1. Настройка проекта с npm, Babel и Webpack.
2. Создание файла HTML.

3. Создание компонентов.
4. Запуск проекта.

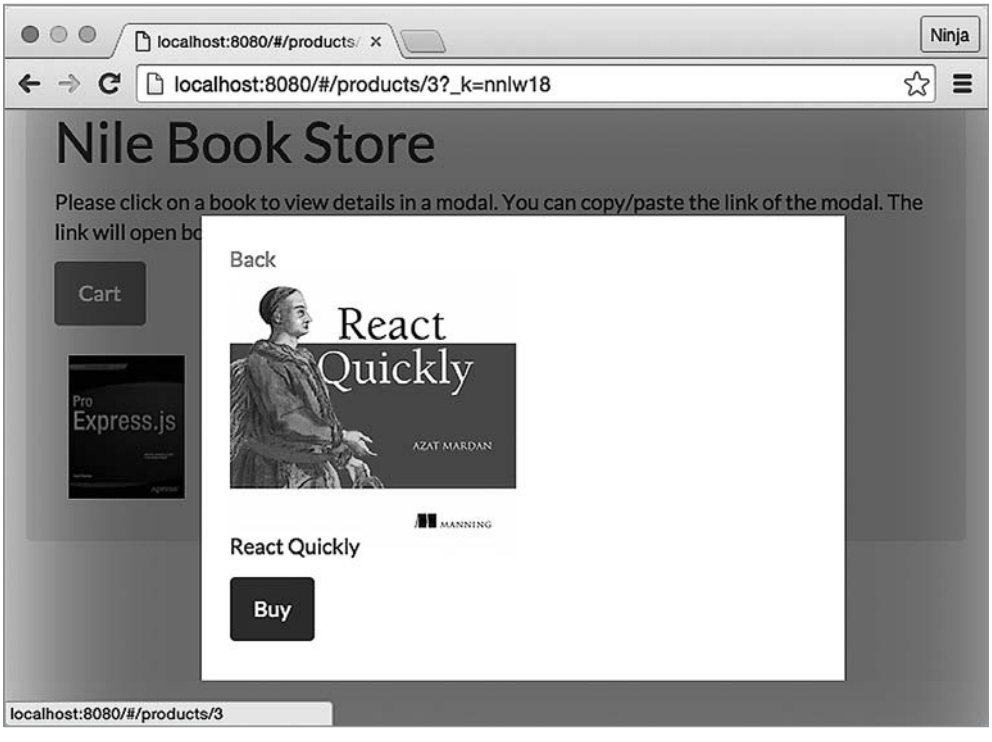


Рис. 18.2. Представление продукта в модальном окне книжного магазина Nile

Я предлагаю вам реализовать задачи, перечисленные в разделе «Домашнее задание» в конце этой главы, и отправить свой код в репозиторий GitHub этой книги: https://github.com/azat-co/react-quickly.

ПРИМЕЧАНИЕ Чтобы повторить приведенное описание, необходимо загрузить неминифицированную версию React, а также установить node.js и npm для компиляции JSX. Я также использую Webpack для сборки. Установка всех этих продуктов рассматривается в приложении А.

ПРИМЕЧАНИЕ Исходный код примеров этой главы доступен по адресам www.manning.com/books/react-quickly и https://github.com/azat-co/react-quickly/tree/master/ch18. Также некоторые демонстрационные примеры доступны по адресу http://reactquickly.co/demos.

Начнем с настройки проекта.

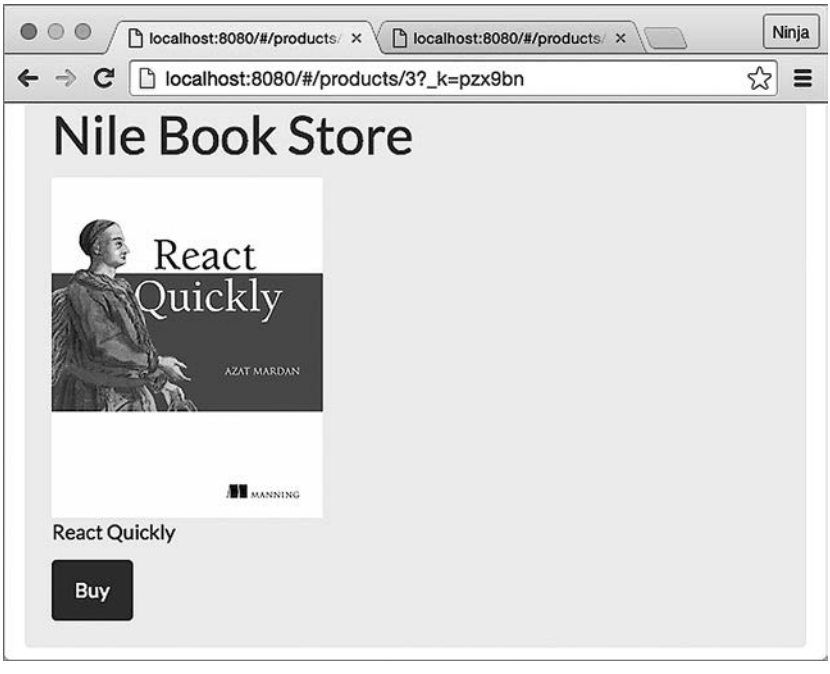


Рис. 18.3. Прямая ссылка открывает представление продукта в новом окне (вместо модального)

18.1. Структура проекта и конфигурация Webpack

Вы уже примерно представляете конечный результат этого проекта: веб-приложение клиентской части с маршрутизацией URL. Структура папок проекта выглядит примерно так:

```
/css
  bootstrap.css
/images
  ...
/js
  bundle.js  <-- Скомпилированный и собранный код
  bundle.js.map
/jsx
  app.jsx  <-- Сценарий точки входа с App и ReactDOM.render()
  cart.jsx  <-- Компонент покупательской корзины
  checkout.jsx
  modal.jsx  <-- Компонент Modal
  product.jsx
/node_modules
```

```
  ...
index.html  <-- Основной файл HTML
package.json
webpack.config.js
```

Для краткости я не привожу содержимое папок images и node_modules. Это чистое приложение клиентской части, но вам понадобится файл package.json для установки зависимостей и передачи инструкций Babel. В листинге 18.1 приведен полный список этих зависимостей из package.json.

Листинг 18.1. Зависимости и конфигурация проекта Nile Book Store

```
{
  "name": "nile",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "author": "Azat Mardan",
  "license": "MIT",
  "scripts": {
    "build": "node ./node_modules/webpack/bin/webpack.js -w"
  },
  "babel": {
    "plugins": [
      "transform-react-jsx"
    ],
    "presets": [
      "es2015"
    ],
    "ignore": [
      "js/bundle.js",
      "node_modules/**/*.js"
    ]
  },
  "devDependencies": {
    "babel-core": "6.3.21",
    "babel-loader": "6.4.1",
    "babel-plugin-transform-react-jsx": "6.3.13",
    "babel-preset-es2015": "6.3.13",
    "history": "4.0.0",
    "react": "15.5.4",
    "react-addons-test-utils": "15.2.1",
    "react-dom": "15.5.4",
    "react-router": "2.8.0",
    "webpack": "2.4.1",
    "webpack-dev-server": "1.14.0"
  }
}
```

Создает сценарий npm для построения ассетов в режиме отслеживания изменений

Добавляет плагин JSX для Babel

Добавляет преобразование ES6/ES2015-в-ES5 (для поддержки старых браузеров)

Исключает зависимости из Babel

Устанавливает библиотеку history для использования с React Router

После запуска со стандартными свойствами проекта команда `scripts` ссылается на локальную установку Webpack. В этом случае вы используете ту же версию, как

в свойстве devDependencies. Сборка создает файл bundle.js и запускает сервер разработки Webpack на порте 8080. Вам не нужно его использовать; вместо этого вы можете производить сборку вручную при каждом внесении изменений и использовать node-static (https://github.com/cloudhead/node-static) или другой локальный веб-сервер:

```
"scripts": {
    "build": "node ./node_modules/webpack/bin/webpack.js -w"
  },
```

Следующая строка необходима для Babel версии 6.x — без нее Babel мало что сделает. Вы говорите Babel использовать преобразователь JSX и конфигурацию ES2015:

```
"babel": {
  "plugins": [
    "transform-react-jsx"
  ],
  "presets": [
    "es2015"
  ],
```

Следующая конфигурация Babel является обязательной. Она исключает из загрузчика Babel некоторые файлы и папки node_modules:

```
  "ignore": [
    "js/bundle.js",
    "node_modules/**/*.js"
  ]
},
```

ПРИМЕЧАНИЕ Затем определяются зависимости. Вы должны использовать именно те номера версий, которые приведены здесь, потому что я не могу гарантировать, что будущие версии станут работать. С учетом скорости развития React и Babel наверняка появятся изменения. Тем не менее нет ничего плохого в том, чтобы использовать чуть более старые версии для изучения концепций, как делается в этой книге.

Зависимости devDependencies предназначены для разработки, на что указывает имя, и не являются частью среды реальной эксплуатации. Здесь указываются Webpack, Webpack Dev Server, Babel и другие пакеты. Пожалуйста, убедитесь в том, что вы используете именно те версии, которые указаны в листинге:

```
  ...
  "devDependencies": {
    "babel-core": "6.3.21",
    "babel-loader": "6.4.1",
    "babel-plugin-transform-react-jsx": "6.3.13",
    "babel-preset-es2015": "6.3.13",
    "history": "4.0.0",
    "react": "15.5.4",
    "react-addons-test-utils": "15.2.1",
```

```
    "react-dom": "15.5.4",
    "react-router": "2.8.0",
    "webpack": "2.4.1",
    "webpack-dev-server": "1.14.0"
  }
}
```

После определения зависимостей проекта необходимо настроить процесс сборки Webpack, чтобы вы могли использовать ES6 и выполнить преобразование JSX. Для этого создайте в корневом каталоге файл webpack.config.js, содержащий следующий код (ch18/nile/webpack.config.js).

Листинг 18.2. Конфигурация Webpack для магазина Nile

```
module.exports = {
  entry: "./jsx/app.jsx",
  output: {
    path: __dirname + '/js',
    filename: "bundle.js"
  },
  devtool: '#sourcemap',
  stats: {
    colors: true,
    reasons: true
  },
  module: {
    loaders: [
      {
        test: /\.jsx?$/,
        exclude: /(node_modules)/,
        loader: 'babel-loader'
      }
    ]
  }
}
```

Выполните команду `npm i` (сокращение от `npm install`), и настройку можно считать завершенной. Затем будет создан файл HTML, содержащий структурные элементы `<div>` для компонентов React.

18.2. Основной файл HTML

Разметка HTML для этого проекта чрезвычайно проста. Она содержит контейнер с идентификатором `content`, а также включает js/bundle.js (ch18/nile/index.html).

Листинг 18.3. Основной файл HTML

```
<!DOCTYPE html>
<html>
  <head>
```

```
    <link href="css/bootstrap.css" type="text/css" rel="stylesheet"/>
  </head>
  <body>
    <div class="container-fluid">
      <div id="content" class=""></div>
    </div>
    <script src="js/bundle.js"></script>
  </body>
</html>
```

Теперь можно быстро проверить, будут ли работать процессы сборки и разработки:

1. Установите все зависимости командой `$ npm install`. Это делается только один раз.
2. Включите команду `console.log('Hey Nile!')` в jsx/app.jsx.
3. Запустите приложение командой `$ npm run build`. Оставьте его работать, потому что ключ `-w` обеспечивает повторную сборку файла при обнаружении изменений.
4. Запустите локальный веб-сервер из корневого каталога проекта. Используйте `node-static` или `webpack-dev-server`, включенный в package.json.
5. Откройте в браузере адрес http://localhost:8080.
6. Откройте консоль браузера (например, Chrome DevTools). На ней должно быть выведено сообщение «Hey Nile!».

18.3. Создание компонентов

Если вы видите сообщение, можно переходить к построению приложения. Начнем с импортирования модулей с использованием синтаксиса модулей ES6 и деструктуризации. Попросту говоря, деструктуризация является способом определения переменной на основе объекта с использованием имени одного из свойств объекта. Например, если вы хотите импортировать `accounts` из `user.accounts` и объявить `accounts` (заметили повторение?), используйте синтаксис `{accounts} = user`. Если вы недостаточно уверенно чувствуете себя с деструктуризацией, обращайтесь к краткой сводке ES6 в приложении Д.

18.3.1. Главный файл: app.jsx

Начнем с файла app.jsx, в котором настраивается основное импортирование, информация о книгах и маршруты. Если не считать кода компонента, до которого мы скоро доберемся, app.jsx выглядит примерно так, как показано в листинге 18.4 (ch18/nile/jsx/app.jsx).

Листинг 18.4. Главный файл приложения

```
const React = require('react')
const ReactDOM = require('react-dom')
const { hashHistory,  <-- Импортирует историю идентификатора фрагмента
  Router,
  Route,
  IndexRoute,
  Link,
  IndexLink
} = require('react-router')  <-- Импортирует объекты из React Router

const Modal = require('./modal.jsx')
const Cart = require('./cart.jsx')
const Checkout = require('./checkout.jsx')
const Product = require('./product.jsx')
```

Импортирует компоненты

```
const PRODUCTS = [
  { id: 0, src: 'images/proexpress-cover.jpg',
    title: 'Pro Express.js', url: 'http://amzn.to/1D6qiqk' },
  { id: 1, src: 'images/practicalnode-cover.jpeg',
    title: 'Practical Node.js', url: 'http://amzn.to/NuQ0fM' },
  { id: 2, src: 'images/expressapiref-cover.jpg',
    title: 'Express API Reference', url: 'http://amzn.to/1xcHanf' },
  { id: 3, src: 'images/reactquickly-cover.jpg',
    title: 'React Quickly',
    url: 'https://www.manning.com/books/react-quickly'},
  { id: 4, src: 'images/fullstack-cover.png',
    title: 'Full Stack JavaScript',
    url: 'http://www.apress.com/9781484217504'}
]
```

Небольшой массив данных книг, чтобы обойтись без работы с базой данных в этом примере

```
const Heading = () => {
  return <h1>Nile Book Store</h1>
}

const Copy = () => {
  return <p>Please click on a book to view details in a modal. You can
  ➥ copy/paste the link of the modal. The link will open the book on a
  ➥ separate page.</p>
}
```

Оба компонента реализованы без состояния

```
class App extends React.Component {
  ...
}

class Index extends React.Component {
  ...
}

let cartItems = {}
const addToCart = (id) => {
  if (cartItems[id])
    cartItems[id] += 1
  else
    cartItems[id] = 1
}
```

Объект cartItems содержит текущие позиции покупательской корзины. В исходном состоянии он пуст

```
ReactDOM.render((
  <Router history={hashHistory}>
    <Route path="/" component={App}>
      <IndexRoute component={Index}/>
      <Route path="/products/:id" component={Product}
        addToCart={addToCart}
        products={PRODUCTS} />
      <Route path="/cart" component={Cart}
      cartItems={cartItems} products={PRODUCTS}/>
    </Route>
    <Route path="/checkout" component={Checkout}
      cartItems={cartItems} products={PRODUCTS}/>
  </Router>
), document.getElementById('content'))
```

После импортирования всего необходимого в начале файла продукты жестко фиксируются в массиве; каждый объект обладает свойствами `id`, `src`, `title` и `url`. Разумеется, в реальном приложении эти данные будут загружаться с сервера, а не храниться в браузерном файле JavaScript:

```
const PRODUCTS = [
  { id: 0, src: 'images/proexpress-cover.jpg',
    title: 'Pro Express.js', url: 'http://amzn.to/1D6qiqk' },
  { id: 1, src: 'images/practicalnode-cover.jpeg',
    title: 'Practical Node.js', url: 'http://amzn.to/NuQ0fM' },
  { id: 2, src: 'images/expressapiref-cover.jpg',
    title: 'Express API Reference', url: 'http://amzn.to/1xcHanf' },
  { id: 3, src: 'images/reactquickly-cover.jpg',
    title: 'React Quickly',
    url: 'https://www.manning.com/books/react-quickly'},
  { id: 4, src: 'images/fullstack-cover.png',
    title: 'Full Stack JavaScript',
    url: 'http://www.apress.com/9781484217504'}
]
```

Следующий компонент реализуется как компонент без состояния с использованием «стрелочного» синтаксиса ES6. Почему не оформить его в виде `<h1>` в разметке? Потому что при таком подходе его можно использовать на нескольких разных экранах. Для `Copy` используется тот же стиль без состояния. Это просто статическая разметка HTML, поэтому вам не понадобится ничего дополнительного, даже свойства:

```
const Heading = () => {
  return <h1>Nile Book Store</h1>
}

const Copy = () => {
  return <p>Please click on a book to view details in a modal. You can
    ➥ copy/paste the link of the modal. The link will open the book on a
    ➥ separate page.</p>
}
```

Далее идут два главных компонента `App` и `Index`, за которыми следует объект `cartItems` с текущим содержимым корзины. Изначально корзина пуста. `addToCart()` — простая функция; в версии на стороне сервера вы используете Redux для сохранения на сервере данных и сеансов, чтобы пользователь мог вернуться к своей корзине позднее:

```
let cartItems = {}
const addToCart = (id) => {
  if (cartItems[id])
    cartItems[id] += 1
  else
    cartItems[id] = 1
}
```

Наконец, метод `ReactDOM.render()` используется для подключения компонента `Router`. React Router необходимо передать библиотеку истории. Как упоминалось ранее, это может быть история браузера или история идентификатора фрагмента (в этом проекте используется второй вариант):

```
ReactDOM.render((
  <Router history={hashHistory}>
    <Route path="/" component={App}>
      <IndexRoute component={Index}/>
      <Route path="/products/:id" component={Product}
        addToCart={addToCart}
        products={PRODUCTS} />
      <Route path="/cart" component={Cart}
        cartItems={cartItems} products={PRODUCTS}/>
    </Route>
    <Route path="/checkout" component={Checkout}
      cartItems={cartItems} products={PRODUCTS}/>
  </Router>
), document.getElementById('content'))
```

Использует компонент Index в IndexRoute

Передает метод для добавления книги в корзину

Передает список книг в корзине и список всех продуктов в cartItems и свойствах продукта

Определяет Checkout за пределами App, чтобы заголовок не отображался

Для маршрута `/products/:id` маршрут компонента `Product` получает функцию `addToCart()`, упрощающую покупку книги. Функция будет доступна в `this.props.route.addToCart`, потому что любое свойство, переданное `Route`, будет доступно в `this.props.route.NAME` в компоненте. Например, `products` преобразуется в `this.props.route.products` в `Product`:

```
<Route path="/products/:id" component={Product} addToCart={addToCart}
  products={PRODUCTS} />
```

Маршрут `/checkout` находится за пределами `App`, поэтому он не содержит заголовка (рис. 18.4). Напомню, что `path` и структура маршрута могут не зависеть друг от друга:

```
<Route path="/checkout" component={Checkout}
  cartItems={cartItems} products={PRODUCTS}/>
```

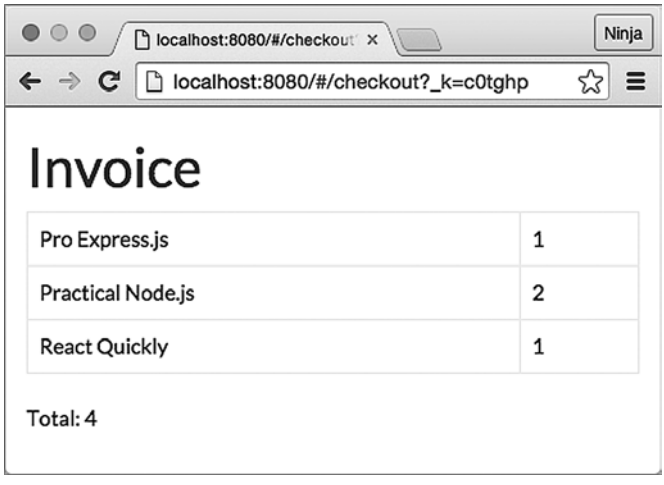


Рис. 18.4. В счете не должно быть заголовка, который отображается в других представлениях

В данном случае, так как `Checkout` размещается вне `App`, `Checkout` не является дочерним элементом `App`. Кнопка Back выполняет переход обратно к приложению с экрана счета/оформления заказа.

Компонент App

А теперь можно реализовать компонент `App`! Это главный компонент, потому что он является точкой входа для Webpack и потому что он предоставляет макет для большинства представлений, рендерит дочерние компоненты, такие как `Product`, список продуктов и `Cart`, а также отображает модальное диалоговое окно. Помните `ReactDOM.render()`? Ниже приведен важнейший фрагмент, который показывает, что `App` является корневым компонентом приложения:

```
ReactDOM.render((
  <Router history={hashHistory}>
    <Route path="/" component={App}>  ◄——  App — предок Product, Cart и Index
      <IndexRoute component={Index}/>
      <Route path="/products/:id" component={Product} .../>
      <Route path="/cart" component={Cart} .../>
    </Route>
    // ...
  </Router>
), document.getElementById('content'))
```

В отличие от компонентов без состояния, которые были обычными функциями, здесь перед нами уже полноценный компонент (ch18/nile/jsx/app.jsx).

Листинг 18.5. Компонент App

```
class App extends React.Component {
  componentWillReceiveProps(nextProps) {
    this.isModal = (nextProps.location.state &&
      nextProps.location.state.modal)
    if (this.isModal &&
      nextProps.location.key !== this.props.location.key) {
      this.previousChildren = this.props.children
    }
  }
  render() {
    console.log('Modal: ', this.isModal)
    return (
      <div className="well">
        <Heading/>
        <div>
          {(this.isModal) ? this.previousChildren :
            this.props.children}

          {(this.isModal)?
            <Modal isOpen={true} returnTo=
              {this.props.location.state.returnTo}>
              {this.props.children}
            </Modal> : ''
          }
        </div>
      </div>
    )
  }
}
```

Использует состояние, переданное в Link (реализуется в Route)

Сохраняет дочерние элементы в previousChildren для рендера

Выводит модальное окно с информацией о книге

Выводит контент старых дочерних элементов (домашняя страница) для модального режима; в противном случае выводит дочерние элементы, определенные в структуре Router

Вспомните, что `componentWillReceiveProps()` получает свойства в аргументе. Этот метод хорошо подходит для проверки модальности представления:

```
class App extends React.Component {
  componentWillReceiveProps(nextProps) {
    this.isModal = (nextProps.location.state &&
      nextProps.location.state.modal)
```

Следующее условие проверяет, является ли текущим модальное или немодальное представление. Если используется модальное представление, то дочерние элементы присваиваются как предыдущие дочерние элементы. Логическая переменная `isModal` проверяет модальность по значению поля `state`, которое берется из свойства `location`, заданного в элементе `Link` (вы увидите пример в компоненте `Index`):

```
  if (this.isModal &&
    nextProps.location.key !== this.props.location.key) {
    this.previousChildren = this.props.children
  }
}
```

Обратите внимание: в функции `render()` неважно, является ли `Heading` обычной функцией (компонентом без состояния). Рендер этого компонента выполняется точно так же, как и рендер любого компонента React:

```
render() {
  console.log('Modal: ', this.isModal)
  return (
    <div className="well">
      <Heading/>
```

Тернарное выражение рендерит либо `this.previousChildren`, либо `this.props.children`. React Router заполняет `this.props.children` по данным других вложенных маршрутов/компонентов, таких как `Index` и `Product`. Напомню, что `App` используется почти всеми экранами приложения. По умолчанию значение `this.props.children` должно рендериться при работе с React Router:

```
<div>
  {(this.isModal) ? this.previousChildren: this.props.children}
```

Если бы проверки `isModal` не было, а вы бы всегда выводили `this.props.children`, то при щелчке на изображении книги для открытия модального окна вы всегда видели бы тот же контент (рис. 18.5). Разумеется, это поведение — совсем не то, что предполагалось. По этой причине выполняется рендер предыдущих дочерних элементов, что в случае модального окна соответствует домашней странице. Вы можете повторно использовать модальную ссылку со значением `state.modal`, равным `true` (см. далее в компоненте `Index`). В результате модальное окно будет отображаться поверх текущего контекста.

Остается отрендерить модальное окно в другом тернарном выражении. Передаются значения `isOpen` и `returnTo`:

```
          {(isModal)?
            <Modal isOpen={true} returnTo={this.props.location.state.returnTo}>
              {this.props.children}
            </Modal> : ''
          }
        </div>
      </div>
    )
  }
}
```

Компонент Index

Продолжим работу над nile/jsx/app.jsx; следующим компонентом станет домашняя страница. Напомню, что на ней выводится полный список книг. Код приводится в листинге 18.6 (ch18/nile/jsx/app.jsx).

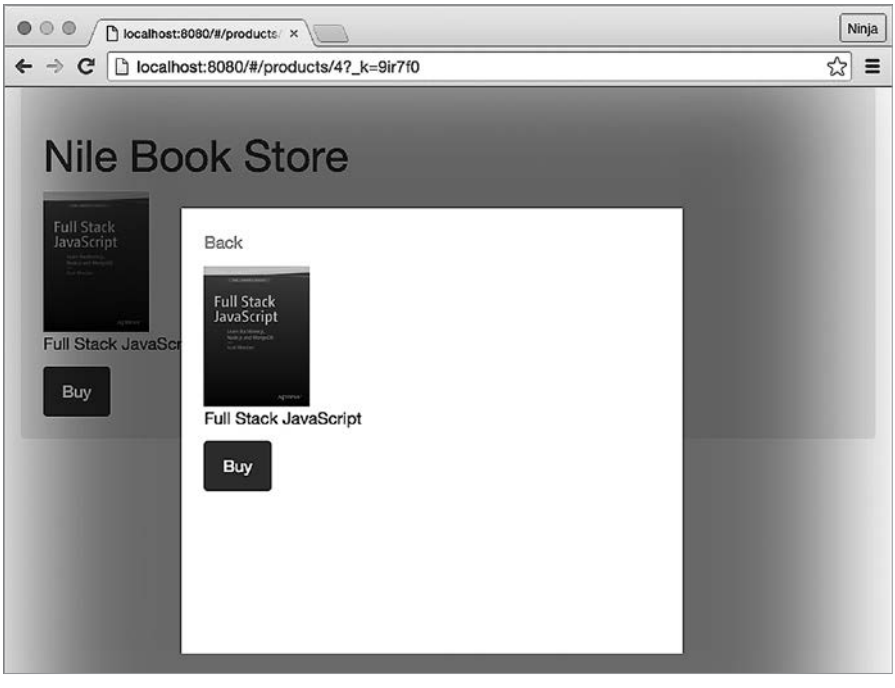


Рис. 18.5. Если не проверить isModal и использовать previousChildren, список книг не выводится

Листинг 18.6. Компонент Index для домашней страницы

```
class Index extends React.Component {
  render() {
    return (
      <div>
        <Copy/>
        <p><Link to="/cart" className="btn btn-danger">Cart</Link></p>
        <div>
          {PRODUCTS.map(picture => (
            <Link key={picture.id}
              to={{pathname: `/products/${picture.id}`,
                state: { modal: true,
                  returnTo: this.props.location.pathname }
                }
              }>
              <img style={{ margin: 10 }} src={picture.src} height="100" />
            </Link>
          ))}
        </div>
      </div>
    )
  }
}
```

Добавляет ссылку на корзину

Использует интерполяцию ES6 (строковый шаблон) для создания URL продукта

Отображает модальное окно

В итераторе `map()` генерируются ссылки на модальные представления окон. При прямом переходе эти ссылки открываются в отдельном немодальном представлении:

```
{PRODUCTS.map(picture => (
    <Link key={picture.id}
      to={{pathname: `/products/${picture.id}`,
        state: { modal: true,
          returnTo: this.props.location.pathname }
        }
      }>
```

Вы можете передать любое свойство компоненту, связанному с маршрутом `/products/:id` (то есть `Product` и его родителю `App`). Для обращения к свойству используется синтаксис `this.props.location.NAME`, где `NAME` — имя свойства. Значение `state.modal` использовалось ранее в компоненте `Modal`.

Тег `<img>` использует атрибут `src` для вывода изображения книги:

```
              <img style={{ margin: 10 }} src={picture.src} height="100" />
            </Link>
          ))}
        </div>
      </div>
    )
  }
}
```

Вот и все, что следовало сказать о файле app.jsx. Следующим будет реализован компонент `Cart`; он будет располагаться в отдельном файле, потому что он не имеет такой тесной связи с приложением, как компонент `App`, который определяет макет книжного магазина.

18.3.2. Компонент Cart

Маршрут `/cart`, рендером которого занимается компонент `Cart`, выводит список книг, помещенных в корзину, и их количество (рис. 18.6). Компонент `Cart` использует `cartItems` для получения списка книг и количества экземпляров. Обратите внимание на применение стиля ES6 в функции `render()` (nile/jsx/cart.jsx).

Листинг 18.7. Компонент Cart

```
const React = require('react')
const {
  Link
} = require('react-router')

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

```
    return <div>
      {(Object.keys(this.props.route.cartItems).length == 0) ?
        <p>Your cart is empty</p> : ''
      }
      <ul>
        {Object.keys(this.props.route.cartItems).map((item,
          index,
          list)=>{
          return <li key={item}>
            {this.props.route.products[item].title}
            - {this.props.route.cartItems[item]}
          </li>
        })}
      </ul>
      <Link to="/checkout"
        className="btn btn-primary">
        Checkout
      </Link>
      <Link to="/" className="btn btn-info">
        Home
      </Link>
    </div>
  }
}

module.exports = Cart
```

Перебирает содержимое списка и рендерит каждый элемент в корзине

Добавляет ссылку для перехода к оформлению заказа, где выводится готовый к печати счет

Добавляет ссылку для перехода к магазину, где пользователь может продолжить покупки

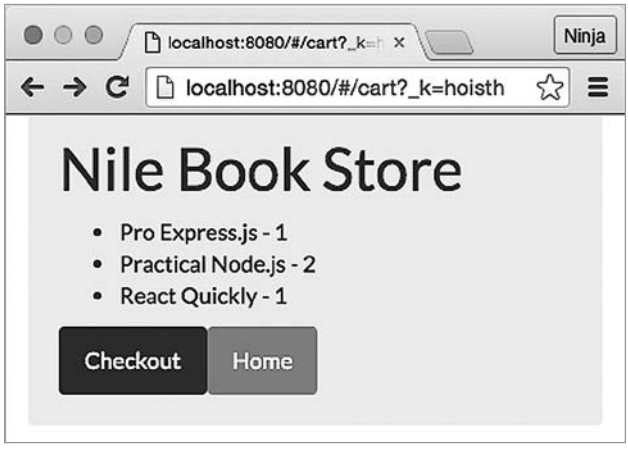


Рис. 18.6. Корзина

`Cart` использует свойство `this.props.route.products`, содержащее список продуктов. Этот возможно, потому что в app.js было определено свойство `route`:

```
<Route path="/cart" component={Cart}
  cartItems={cartItems} products={PRODUCTS}/>
```

При использовании Redux (глава 14) вам не придется вручную передавать такие свойства, как `products`, потому что `Provider` будет автоматически заполнять хранилище данных в дочерних компонентах.

18.3.3. Компонент Checkout

Далее идет компонент `Checkout`, показанный на рис. 18.7. Это единственный компонент, не входящий в маршрут `App`. Напомню, как выглядит маршрутизация из app.js:

```
ReactDOM.render((
  <Router history={hashHistory}>
    <Route path="/" component={App}>  <-- Маршрут App: основной макет
      <IndexRoute component={Index}/>
      <Route path="/products/:id" component={Product}
        addToCart={addToCart}
        products={PRODUCTS} />
      <Route path="/cart" component={Cart}
        cartItems={cartItems} products={PRODUCTS}/>
    </Route>
    <Route path="/checkout" component={Checkout}  <-- Маршрут Checkout лежит
      cartItems={cartItems} products={PRODUCTS}/>     вне маршрута App
  </Router>
), document.getElementById('content'))
```

Как видите, `App` и `Checkout` находятся на одном уровне иерархии. Таким образом, при переходе к `/checkout` маршрут `App` не сработает. Макет отсутствует. (Интересно, что при вложении URL компоненты могут не участвовать во вложенной структуре: например, `/cart/checkout`. Впрочем, здесь мы так поступать не будем.)

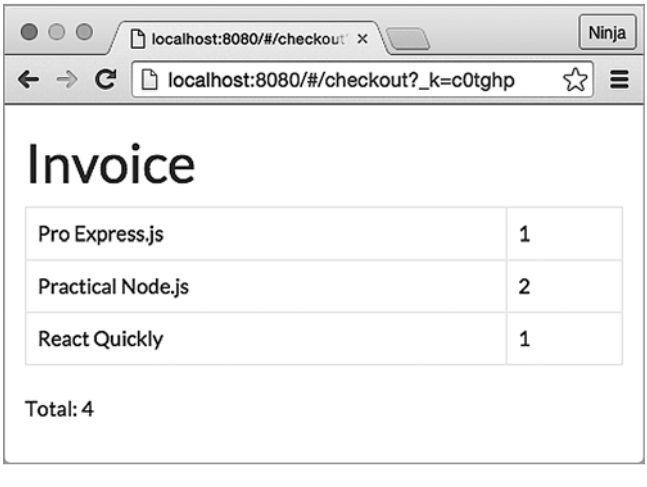


Рис. 18.7. Компоненту Checkout заголовок не нужен

Счет, подготовленный для печати, использует таблицу Twitter Bootstrap и стили `table-bordered`. И снова мы используем возможности ES6: `const` (помните, свойства объекта могут изменяться) и синтаксис функции (nile/jsx/checkout.jsx).

Листинг 18.8. Компонент Checkout

```
const React = require('react')
const {
  Link
} = require('react-router')

class Checkout extends React.Component {
  render() {
    let count = 0
    return <div><h1>Invoice</h1><table className="table table-bordered">
      <tbody>
      {Object.keys(this.props.route.cartItems).map((item, index,
        list)=>{  <-- Перебирает содержимое списка и рендерит каждый элемент в корзине
        count += this.props.route.cartItems[item]
        return <tr key={item}>
          <td>{this.props.route.products[item].title}</td>  <-- Использует список продуктов, переданных в маршруте, для вывода заголовка
          <td>{this.props.route.cartItems[item]}</td>
        </tr>
      })}
    </tbody></table><p>Total: {count}</p></div>
  }
}

module.exports = Checkout  <-- Экспортирует класс
```

Теперь необходимо реализовать компонент `Modal`.

18.3.4. Компонент Modal

Этот компонент выводит свои дочерние элементы в модальном диалоговом окне. Вспомните, что в коде `App` компонент `Modal` использовался следующим образом:

```
{(this.isModal) ?
<Modal isOpen={true} returnTo={this.props.location.state.returnTo}>
{this.props.children}
</Modal> : ''
}
```

`Modal` берет свои дочерние элементы из свойства `this.props.children` компонента `App`, который, в свою очередь, определяется в `app.js` внутри `<Route>`. Напомню структуру маршрутизации:

```
ReactDOM.render((
  <Router history={hashHistory}>
    <Route path="/" component={App}>
      <IndexRoute component={Index}/>
```

```
      <Route path="/products/:id" component={Product}
        addToCart={addToCart}
        products={PRODUCTS} />
      <Route path="/cart" component={Cart}
        cartItems={cartItems} products={PRODUCTS}/>
    </Route>
    <Route path="/checkout" component={Checkout}
      cartItems={cartItems} products={PRODUCTS}/>
  </Router>
), document.getElementById('content'))
```

Так реализуется возможность страницы продукта как в нормальном, так и в модальном режиме. Компоненты, вложенные в маршрут App, являются его дочерними элементами, зависящими от URL (nile/jsx/modal.jsx).

Листинг 18.9. Компонент Modal

```
const React = require('react')
const {
  Link
} = require('react-router')

class Modal extends React.Component {
  constructor(props) {
    super(props)
    this.styles = {
      position: 'fixed',
      top: '20%',
      right: '20%',
      bottom: '20%',
      left: '20%',
      width: 450,
      height: 400,
      padding: 20,
      boxShadow: '0px 0px 150px 130px rgba(0, 0, 0, 0.5)',
      overflow: 'auto',
      background: '#fff'
    }
  }
  render() {
    return (
      <div style={this.styles}>
        <p>
          <Link to={this.props.returnTo}>
            Back
          </Link>
        </p>
        {this.props.children}
      </div>
    )
  }
}

module.exports = Modal
```

Определяет стили как атрибуты экземпляра класса

Использует фиксированную позицию (в сочетании с top, right, left и bottom) для размещения модального диалогового окна в середине

Обратите внимание на «верблюжий регистр» в имени boxShadow: в CSS это box-shadow

Применяет стили для создания модального представления

В модальном окне выводится компонент `Product`, потому что он вложен в `App` в маршрутизации, а маршрут `Product` использует URL `/product/:id`, который использовался наряду с истинным значением `set.modal` в `Index` (список продуктов).

18.3.5. Компонент Product

Компонент `Product` использует свойство из своего маршрута для инициирования действий (`this.props.route.addToCart`). Метод `addToCart()` в app.js помещает конкретную книгу в корзину (если вы используете Redux, при этом происходит передача действия). Вызов `addToCart()` инициируется в обработчике события `onClick` браузера и в локальном методе `handleBuy()` в `Product`, который вызывает `addToCart` из app.js. Подведем итог: `onClick`→`this.handleBuy`→`this.props.route.addToCart`→`addToCart()` (app.js). Напомню, что метод `addToCart()` выглядит так:

```
let cartItems = {}
const addToCart = (id) => {
  if (cartItems[id])
    cartItems[id] += 1
  else
    cartItems[id] = 1
}
```

Конечно, если бы вы применяли Redux или Relay, то использовали бы их методы. В этом примере для простоты используется простой массив, выполняющий функции хранилища данных, и один метод.

Теперь рассмотрим сам компонент `Product`. Как обычно, все начинается с импортирования React и определения класса; затем происходит обработка события и рендер. В листинге 18.10 приведен полный код `Product` (nile/jsx/product.jsx) с пометкой самых интересных частей.

Листинг 18.10. Информация об отдельном продукте

```
const React = require('react')
const {
  Link
} = require('react-router')

class Product extends React.Component {
  constructor(props) {
    super(props)
    this.handleBuy = this.handleBuy.bind(this)
  }
  handleBuy (event) {
    this.props.route.addToCart(this.props.params.id)
  }
  render() {
    return (
```

Выполняет связывание для того, чтобы использовалось правильное значение this

Передает идентификатор книги функции в app.jsx

```
      <div>
        <img src={this.props.route.products[
        ➥ this.props.params.id].src}
          style={{ height: '80%' }} />
        <p>{this.props.route.products[this.props.params.id].title}</p>
        <Link
          to={{
            pathname: `/cart`,
            state: { productId: this.props.params.id}
          }}
          onClick={this.handleBuy}
          className="btn btn-primary">
            Buy
        </Link>
      </div>
    )
  }
}

module.exports = Product
```

Обеспечивает вызов функции при щелчке на кнопке Buy

Также можно отправить состояние `Cart` в компоненте `Link`:

```
<Link
  to={{
    pathname: `/cart`,
    state: { productId: this.props.params.id}
  }}
  onClick={this.handleBuy}
  className="btn btn-primary">
    Buy
</Link>
```

Напомню, что `Product` используется `Modal` косвенно: `Modal` не выполняет рендер `Product`. Вместо этого `Modal` использует свойство `this.props.children`, содержащее `Product`. Таким образом, `Modal` можно считать *сквозным* (passthrough) компонентом. (За дополнительной информацией о свойстве `this.props.children` и сквозных компонентах, использующих его, обращайтесь к главе 8.)

18.4. Запуск проекта

Работа над проектом книжного магазина завершена. Вы использовали некоторые возможности ES6 и передавали состояния с использованием React Router. Чтобы запустить проект, постройте его командой `npm run build`, запустите локальный веб-сервер (WDS или node-static) и перейдите по адресу http://localhost:8080/nile; предполагается, что статический веб-сервер работает в родительской папке, содержащей папку nile (путь в URL зависит от того, где был запущен веб-сервер).

На экране должна появиться домашняя страница с обложками книг. Если щелкнуть на обложке, появляется модальное окно; кнопка Buy добавляет книгу в корзину, содержимое которой отображается на страницах `/cart` и `/checkout`.

18.5. Домашнее задание

Для получения дополнительных баллов сделайте следующее:

- Абстрагируйте (проще говоря, скопируйте/вставьте) `Index` и `App` в разные файлы, отличные от app.js. Переименуйте App в `Layout`.
- Переместите данные в долгосрочное хранилище, например MongoDB или PostgreSQL.
- Преобразуйте URL-адрес так, чтобы идентификатор фрагмента в нем не использовался. Для этого вам понадобится API-истории наряду со специализированным сервером Express (который вам нужно будет реализовать). За идеями обращайтесь к клону Netflix с URL без идентификатора фрагмента в главе 15.
- Добавьте модульные тесты для `Product` и `Checkout` с использованием Jest.

Отправьте свой код в новую папку в ch18 как pull-запрос в репозиторий GitHub этой книги: https://github.com/azat-co/react-quickly.

18.6. Итоги

- Компонент `Link` импортируется из `react-router` и может использоваться для передачи состояния как в примере `<Link to={{pathname: '/product', state: { modal: true }}}>`.
- Состояние React Router доступно в форме `this.props.location.state`.
- Вы можете передавать свойства, определенные в `<Route` *имя*`={`*значение*`}>`, и эти свойства будут доступны в `this.props.route.name`.

19 Проект: проверка паролей с Jest

Посмотрите вступительный видеоролик к этой главе, отсканировав QR-код или перейдя по ссылке http://reactquickly.co/videos/ch19.

ЭТА ГЛАВА ОХВАТЫВАЕТ СЛЕДУЮЩИЕ ТЕМЫ:

- Структура проекта и конфигурация Webpack.
- Основной файл HTML.
- Реализация модуля надежного пароля.
- Создания тестов Jest.
- Реализация компонента Password.

В этом проекте основное внимание будет сосредоточено на построении пользовательского интерфейса, работе с модулями и тестировании с использованием Jest, а также на других приемах, относящихся к React: композиции компонентов, синтаксису ES6, состоянию, свойствам и т. д. Вспомните, что тестирование рассматривалось в главе 16 — мы использовали виджет проверки пароля как пример модульного и UI-тестирования. В этом проекте будет построен сам виджет для проверки и генерирования новых паролей. Попутно я буду снова объяснять тестирование в расширенном формате.

Виджет содержит кнопку Save, которая заблокирована по умолчанию, но становится доступной, если пароль достаточно надежен (согласно заранее заданным правилам), как показано на рис. 19.1. Кроме того, кнопка Generate позволяет создать надежный (согласно критериям) пароль. Все выполняемые правила перечеркиваются. Также имеется флажок Show Password, который скрывает/отображает пароль как во многих интерфейсах macOS (рис. 19.2).

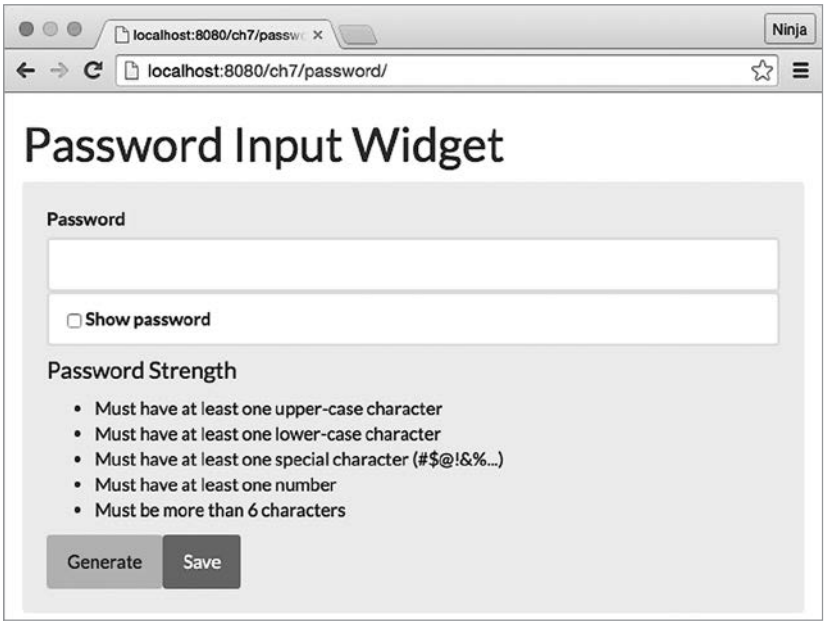


Рис. 19.1. Виджет для ввода пароля, который позволяет ввести пароль или автоматически сгенерировать пароль, удовлетворяющий заданным критериям надежности

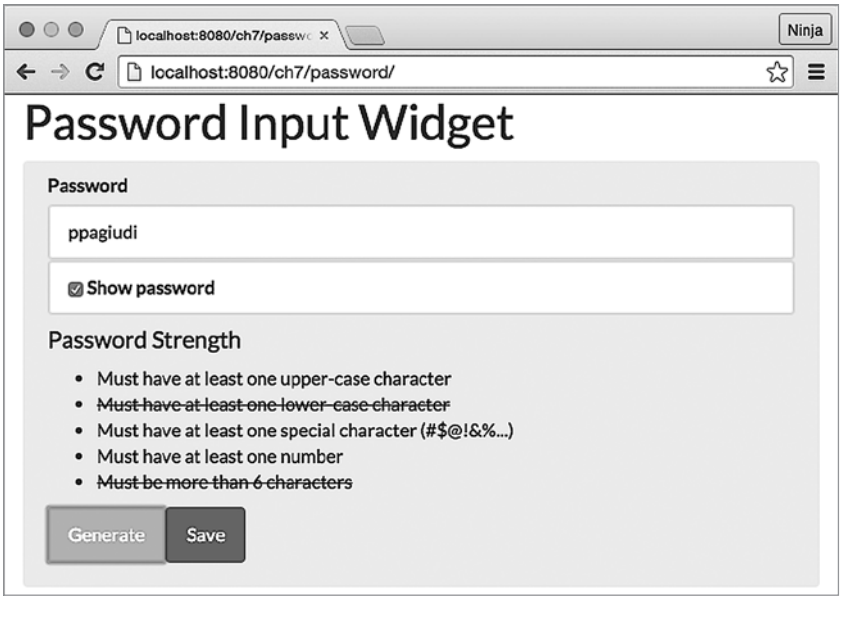


Рис. 19.2. Виджет с частью выполненных критериев и видимым паролем

Родительский компонент называется `Password`, а дочерние компоненты перечислены ниже:

- `PasswordInput` — текстовое поле для ввода пароля.
- `PasswordVisibility` — флажок для переключения видимости пароля.
- `PasswordInfo` — список критериев, которые должны быть выполнены перед сохранением пароля.
- `PasswordGenerate` — кнопка для генерирования пароля, удовлетворяющего всем критериям.

Виджет строится с использованием одного родительского компонента. Правила надежности пароля передаются компоненту в свойствах, так что компонент обладает широкими возможностями настройки. Готов поспорить, что после некоторой доработки вы сможете использовать его в своих приложениях!

ПРИМЕЧАНИЕ Чтобы повторить приведенное описание, необходимо установить Node.js и npm для компиляции JSX. В примере также используется Webpack для сборки и, конечно, Jest как средство тестирования. Установка всех этих продуктов рассматривается в приложении A.

ПРИМЕЧАНИЕ Так как отдельные части проекта были представлены в главе 16, исходный код хранится в папке ch16; он доступен по адресам www.manning.com/books/react-quickly и https://github.com/azat-co/react-quickly/tree/master/ch16. Некоторые демоверсии также доступны по адресу http://reactquickly.co/demos.

Начнем с настройки проекта.

19.1. Структура проекта и конфигурация Webpack

Так выглядит полная структура проекта. Начните с создания новой папки проекта с именем password:

```
/password
  /__tests__  <-- Эта папка содержит все файлы тестов Jest
    generate-password.test.js
    password.test.js
  /css
    bootstrap.css
  /dist  <-- Папка для файлов, собранных Webpack
    bundle.js
    bundle.js.map
  /js
    generate-password.js
```

```
  rules.js
/jsx
  app.jsx  ◄── Точка входа приложения
  password-generate.jsx  ◄── Библиотека, отвечающая за генерирование случайных паролей
  password-info.jsx
  password-input.jsx
  password-visibility.jsx
  password.jsx
/node_modules
  ...
index.html
package.json
README.md
webpack.config.js  ◄── Конфигурационный файл Webpack
```

Папка __tests__ предназначена для тестов Jest. В папке css хранится моя тема Twitter Bootstrap, которая называется Flatly (https://bootswatch.com/flatly). В папках js и jsx находятся библиотеки и компоненты соответственно. В файле js/generate-password.js хранится библиотека, отвечающая за генерирование случайных паролей.

Папка dist содержит откомпилированные файлы JSX с картами исходного кода. Здесь Webpack размещает объединенный файл и его карту исходного кода. Имя dist является сокращением *distribution*, то есть «поставка»: это такое же распространенное имя, как js или build.

Не забудьте: для того, чтобы вам не приходилось устанавливать каждую зависимость с точной версией вручную, вы можете скопировать package.json из листинга 19.1 в папку password, а затем выполнить в ней команду `npm install` (ch16/password/package.json).

Листинг 19.1. Зависимости и настройка конфигурации проекта

```
{
  "name": "password",
  "version": "2.0.0",
  "description": "",
  "main": "index.html",
  "scripts": {
    "test": "jest",
    "test-watch": "jest --watch",
    "build-watch": "./node_modules/.bin/webpack -w",   ◄── Создает сценарий npm для построения ассетов с использованием Webpack и отслеживания изменений
    "build": "./node_modules/.bin/webpack"
  },
  "author": "Azat Mardan",
  "license": "MIT",
  "babel": {
    "presets": [
      "react"  ◄── Использует Babel в Jest для поддержки JSX
    ]
  },
  "devDependencies": {
```

```
    "babel-core": "6.10.4",
    "babel-loader": "6.4.1",
    "babel-preset-react": "6.5.0",
    "jest-cli": "19.0.2",  ◄—— Использует Jest как локальный модуль (рекомендуется)
    "react": "15.5.4",
    "react-test-renderer": "15.5.4",
    "react-dom": "15.5.4",  ◄—— Использует react-test-renderer для неглубокого рендеринга
    "webpack": "2.4.1"
  }
}
```

Здесь интерес представляет секция `scripts`, которая используется для тестирования, компиляции и упаковки:

```
"scripts": {
  "test": "jest",
  "test-watch": "jest --watch",
  "build-watch": "./node_modules/.bin/webpack -w",
  "build": "./node_modules/.bin/webpack"
},
```

Вспомните, что в хранилище Nile в главе 18 использовался плагин `transform-react-jsx`:

```
"babel": {
  "plugins": [
    "transform-react-jsx"
  ],
```

Но в этом проекте используется конфигурация (preset) React — это другой способ достижения той же цели. Вы можете использовать конфигурацию или плагин на свой выбор. Конфигурации считаются более современным решением и чаще используются в документации и проектах.

Тестовый сценарий (`npm test`) предназначен для ручного запуска тестов Jest. И наоборот, сценарий `test-watch` предназначен для выполнения Jest в фоновом режиме. `test-watch` запускается командой `npm run test-watch`, потому что включение `run` не обязательно только для `test` и `start`. Сценарий `test-watch` запускается однократно, а Jest (в режиме отслеживания изменений) замечает любые изменения в исходном коде и запускает тесты заново. Пример вывода:

```
PASS __tests__/password.test.js
  PASS __tests__/generate-password.test.js

Test Suites: 2 passed, 2 total
Tests: 3 passed, 3 total
Snapshots: 0 total
Time: 1.502s
Ran all test suites.
```

```
Watch Usage
 › Press o to only run tests related to changed files.
 › Press p to filter by a filename regex pattern.
 › Press t to filter by a test name regex pattern.
 › Press q to quit watch mode.
 › Press Enter to trigger a test run.
```

К настоящему моменту вы определили зависимости проекта. Следующим шагом станет настройка процесса сборки Webpack, чтобы обеспечить преобразование разметки JSX в JS. Для этого создайте файл webpack.config.js в корневом каталоге с кодом из листинга 19.2 (ch16/password/webpack.config.js).

Листинг 19.2. Конфигурация Webpack

```
module.exports = {
  entry: './jsx/app.jsx',
  output: {
    path: __dirname + '/dist/',
    filename: 'bundle.js'
  },
  devtool: '#sourcemap',
  stats: {
    colors: true,
    reasons: true
  },
  module: {
    loaders: [
      {
        test: /\.jsx?$/,
        exclude: /(node_modules)/,
        loader: 'babel-loader'
      }
    ]
  }
}
```

Назначает точку входа для проекта (таких точек может быть несколько)

Настраивает карты исходного кода для просмотра правильных номеров строк в DevTools

Применяет Babel с использованием конфигураций Babel из package.json

Теперь вы можете определить конфигурации для построения проекта в webpack.config.js. Точка входа находится в JSX-файле app.js, а приемником является папка dist. Кроме того, параметры конфигурации задают загрузчик Babel (для преобразования JSX в JS) и включают карты исходного кода.

Сборка может запускаться командой `./node_modules/.bin/webpack` или `./node_modules/.bin/webpack -w`, если вам нужна функциональность отслеживания изменений в файлах. Да, с ключом `-w` можно заставить Webpack выполнять повторную сборку при каждом изменении файлов — то есть каждый раз, когда вы выбираете команду сохранения в Блокноте (я не люблю интегрированные среды разработки). Режим отслеживания изменений очень удобен для активной разработки!

Вы можете создавать несколько разновидностей файла webpack.config.js и указывать разные имена файлов в параметре `--config`:

```
$ ./node_modules/.bin/webpack --config production.config.js
```

Каждый конфигурационный файл может использовать свой сценарий в package.json для удобства.

Итак, пакет Webpack прост и удобен тем, что он поддерживает модули CommonJS/Node по умолчанию. Вам не понадобятся ни Browserify, ни какие-либо другие загрузчики модулей. С Webpack все выглядит так, словно вы пишете программу Node для браузерного JavaScript!

19.2. Основной файл HTML

Затем создайте файл index.html. Он содержит контейнер с идентификатором `content` и включает dist/bundle.js (ch16/password/index.html).

Листинг 19.3. Основной файл HTML

```
<!DOCTYPE html>
<html>

  <head>
    <link href="css/bootstrap.css" rel="stylesheet" type="text/css"/>
  </head>

  <body class="container">
    <h1>Password Input Widget</h1>
    <div id="password"></div>
    <script src="dist/bundle.js" ></script>  <-- Загружает упакованное приложение
  </body>
</html>
```

Теперь все должно быть настроено и готово к началу разработки. Тестирование в процессе разработки лучше проводить в инкрементном режиме, чтобы область для диагностики ошибок была как можно меньше. Проведите быструю проверку и убедитесь в том, что конфигурация настроена правильно, как это было сделано в главе 18. Действуйте примерно по такой схеме:

1. Установите все зависимости командой `$ npm install`. Это делается только один раз.
2. Включите команду `console.log('Painless JavaScript password!')` в jsx/app.jsx.
3. Запустите приложение командой `npm start`. Оставьте его работать, потому что ключ `-w` обеспечивает повторную сборку файла при обнаружении изменений.
4. Запустите локальный веб-сервер из корневого каталога проекта.
5. Откройте в браузере адрес http://localhost:8080.
6. Откройте консоль браузера (например, Chrome DevTools). На ней должно быть выведено сообщение «Painless JavaScript password!».

19.3. Реализация модуля надежного пароля

Модуль `strong-password` представляет собой файл generate-password.js в папке password/js. Тест для файла хранится в файле password/__tests__/generate-password.test.js. При вызове модуль возвращает случайные пароли, которые содержат комбинацию различных типов символов:

- Специальные символы — `!@\#$%^&*()_+{}:"<>?\|[]\',./`~`.
- Символы нижнего регистра — `abcdefghijklmnopqrstuvwxyz`.
- Символы верхнего регистра — `ABCDEFGHIJKLMNOPQRSTUVWXYZ`.
- Цифры — `0123456789`.

Эти категории наряду с длиной последовательности и случайностью выбора гарантируют надежность пароля. Используя методологию TDD/BDD, начнем с реализации тестов.

19.3.1. Тесты

Начнем с тестов в файле generate-password.test.js. Помните, для того чтобы тесты были автоматически найдены Jest, они должны храниться в папке __tests__(ch16/password/__tests__/generate-password.test.js).

Листинг 19.4. Тесты для модуля паролей

```
const generatePassword = require('../js/generate-password.js')
const pattern = /^[A-Za-z0-9\!\@\#\$\%\^\&\*\(\)\_\+\{\}\:\"\<\>\?\\|\
➥ [\]\/'\,\.\`\~]{8,16}$/

describe('method generatePassword', ()=>{
  let password, password2
  it('returns a generated password of the set pattern', ()=>{
    password = generatePassword()
    expect(password).toMatch(pattern)
  })
  it('return a new value different from the previous one', ()=>{
    password2 = generatePassword()
    expect(password2).toMatch(pattern)
    expect(password2).not.toEqual(password)
  })

})
```

Определяет регулярное выражение для пароля, удовлетворяющего всем критериям

Проверяет, что сгенерированный пароль соответствует шаблону

Проверяет, что при вызове метода возвращается новый пароль

Сначала программа объявляет переменную `password` и импортирует generate-password.js. Регулярное выражение проверяет содержимое и длину пароля. Система не идеальна, потому что мы не проверяем, содержит ли каждый пароль хотя бы один из этих символов, но пока хватит и этого:

```
let password,
  password2,
  pattern = /^[A-Za-z0-9\!\@\#\$\%\^\&\*\(\)\_\+\{\}\:\"\<\>\?\\|
  ➥ \[\]\/'\,\.\`\~]{8,16}$/
```

Включите в набор тестов `describe` новую сущность `method generatePassword`. Она определяет то, что вы собираетесь тестировать: функцию, экспортируемую в модуле generate-password.js.

Реализуйте набор тестов `it` с кодом, к которому должно применяться модульное тестирование командами `expect` в стиле BDD (см. главу 16). Как минимум проверьте пароль по шаблону регулярного выражения:

```
describe('method generatePassword', () => {
  it('returns a generated password of the set pattern', ()=>{
    password = generatePassword()
    expect(password).toMatch(pattern)
  })
  it('returns a new value different from the previous one', ()=>{
    password2 = generatePassword()
    expect(password2).not.toEqual(password)
  })
})
```

А если пароль не будет изменяться каждый раз, когда вы вызываете `generatePassword()`? Что, если он жестко зафиксирован в generate-password.js? Конечно, такого быть не должно! Поэтому второй набор тестов проверяет, отличается ли второй сгенерированный пароль от первого.

19.3.2. Код

Модуль надежного пароля будет реализован в файле js/generate-password.js, чтобы вы могли с ходу правильно применить TDD/BDD, то есть сначала написать тест, а потом перейти к коду. Ниже приведен гибкий генератор паролей, использующий три набора символов для выполнения критериев надежного пароля:

```
const SPECIALS = '!@#$%^&*()_+{}:"<>?\|[]\',./`~'
const LOWERCASE = 'abcdefghijklmnopqrstuvwxyz'
const UPPERCASE = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
const NUMBERS = '0123456789'
const ALL = `${SPECIALS}${LOWERCASE}${UPPERCASE}${NUMBERS}`

const getIterable = (length) => Array.from({length},
  (_, index) => index + 1)

const pick = (set, min, max) => {
  let length = min
  if (typeof max !== 'undefined') {
    length += Math.floor(Math.random() * (max - min))
  }
```

Добавляет +1, чтобы избежать 0 в качестве значения, и использует неявный возврат

Определяет функцию выбора, которая возвращает символы из набора между min и max

```
  return getIterable(length).map(() => (
    set.charAt(Math.floor(Math.random() * set.length))
  )).join('')
}

const shuffle = (set) => {
  let array = set.split('')
  let length = array.length
  let iterable = getIterable(length).reverse()
  let shuffled = iterable.reduce((acc, value, index) => {
    let randomIndex = Math.floor(Math.random() * value)
    [acc[value -1], acc[randomIndex]] = [acc[randomIndex], acc[value - 1]]
    return acc
  }, [...array])
  return shuffled.join('')
}

module.exports = () => {
  let password = (pick(SPECIALS, 1)
    + pick(LOWERCASE, 1)
    + pick(NUMBERS, 1)
    + pick(UPPERCASE, 1)
    + pick(ALL, 4, 12))
  return shuffle(password)
}
```

Создает итерируемый элемент с пустыми строками

Перемешивает символы, чтобы получить случайность

Изменяет итерабельность, чтобы получить значение от max до min

Применяет редьюсер для получения перетасованного массива

Определяет правила для удовлетворения виджетов

Экспортированная функция (присвоенная `module.exports`) вызывает метод `shuffle()`, который выполняет *случайную* перестановку символов в строке. Метод `shuffle()` берет пароль, сгенерированный `pick()`, который использует наборы символов, чтобы сгенерированный пароль включал как минимум один символ из каждой группы (цифры, символы верхнего регистра, специальные символы и т. д.). Последняя часть пароля состоит из более случайных элементов из объединения `ALL`.

Вы можете запустить модульный тест для password/__tests__/generate-password.js командой `jest __tests__/generate-password.test.js` или `npm test __tests__/generate-password.test.js`, выполненной из корневого каталога проекта (папки password). Тест должен пройти и выдать сообщение, которое выглядит приблизительно так:

```
jest __tests__/generate-password.test.js

 PASS __tests__/generate-password.test.js
  method generatePassword
    ✓ returns a generated password of the set pattern (4ms)
    ✓ return a new value different from the previous one (2ms)

Test Suites: 1 passed, 1 total
Tests: 2 passed, 2 total
Snapshots: 0 total
Time: 1.14s
Ran all test suites matching "__tests__/generate-password.test.js".
```

19.4. Реализация компонента Password

Пора сделать следующий логичный шаг — перейти к работе над главным компонентом `Password`. По канонам TDD начинать всегда следует с теста: в данном случае UI-теста, потому что тестироваться должно поведение (например, щелчки на кнопках).

19.4.1. Тесты

Создайте файл UI-теста с именем __tests__/password.test.js. Этот файл уже рассматривался в главе 16, поэтому я приведу здесь полный пример с комментариями (ch16/password/__tests__/password.test.js).

Листинг 19.5. Спецификация компонента Password

```
describe('Password', function() {
  it('changes after clicking the Generate button', (done)=>{
    const TestUtils = require('react-addons-test-utils')
    const React = require('react')
    const ReactDOM = require('react-dom')
    const Password = require('../jsx/password.jsx')

    const PasswordGenerate = require('../jsx/password-generate.jsx')
    const PasswordInfo = require('../jsx/password-info.jsx')
    const PasswordInput = require('../jsx/password-input.jsx')
    const PasswordVisibility = require('../jsx/password-visibility.jsx')

    const fD = ReactDOM.findDOMNode

    let password = TestUtils.renderIntoDocument(<Password
      upperCase={true}
      lowerCase={true}
      special={true}
      number={true}
      over6={true}
      />
    )

    let rules = TestUtils.scryRenderedDOMComponentsWithTag(password,
      'li')
    expect(rules.length).toBe(5)
    expect(fD(rules[0]).textContent).toEqual('Must have
    ➥ at least one upper-case character')
    let generateButton = TestUtils.findRenderedDOMComponentWithClass(password,
    ➥ 'generate-btn')
    expect(fD(rules[1]).firstChild.nodeName.toLowerCase()).
    ➥ toBe('#text')
    TestUtils.Simulate.click(fD(generateButton))
    expect(fD(rules[1]).firstChild.nodeName.toLowerCase()).
    ➥ toBe('strike')
    done()
  })
```

Включает библиотеки

Создает компонент React благодаря поддержке JSX из пакета babel-jest (часть Jest: https://github.com/facebook/jest/tree/master/packages/babel-jest)

Получает элементы списка (<li>)

Проверяет, совпадает ли текст первого элемента <li> с ожидаемым значением

Получает кнопку для генерирования паролей

Проверяет выполнение второго критерия

Моделирует щелчок на Generate

Проверяет, выполняется ли второй критерий — то есть что первый дочерний элемент содержит обычный текст, а не <strike>

Этот тестовый случай можно расширить для проверки всех свойств и правил; вы можете заняться этим самостоятельно (другие идеи описаны в разделе «Домашнее задание» в конце этой главы). Также желательно создать другой набор тестов с другим сочетанием свойств и провести тестирование и для него.

Вот и все! Тест (`npm test` или `jest`) должен завершиться неудачей с выдачей сообщения об ошибке:

```
Error: Cannot find module '../jsx/password.jsx' from 'password.test.js'
```

Такая ситуация нормальна для разработки через тестирование, когда тесты пишутся раньше кода. Теперь необходимо сделать следующий шаг и реализовать компонент `Password`.

19.4.2. Код

На этом шаге мы создадим компонент `Password` с некоторым исходным состоянием. Переменные состояния:

- `strength` — объект с критериями силы (то есть набор правил, каждое из которых равно `true` или `false` в зависимости от того, выполняется ли данный критерий).
- `password` — текущий пароль.
- `visible` — флаг видимости поля ввода пароля.
- `ok` — флаг выполнения всех правил; позволяет пользователю сохранить пароль (снимает блокировку с кнопки Save).

Представьте, что через несколько дней после завершения работы над виджетом разработчик из другой команды хочет использовать ваш компонент с чуть более жесткими критериями надежности. Лучше всего абстрагировать (скопировать/вставить, проще говоря) код с критериями (правилами) надежности пароля в отдельный файл. Это следует сделать до того, как продолжать работу над password.jsx.

Создайте файл с именем rules.js (ch16/password/js/rules.js). Этот файл реализует правила надежности пароля, которые можно использовать в password.jsx для проведения проверки и вывода предупреждений. Отделение правил упростит изменение, добавление и удаление правил в будущем.

Листинг 19.6. Правила надежности пароля

```
module.exports = {
  upperCase: {
    message: 'Must have at least one upper-case character',
    pattern: /([A-Z]+)/
  },
  lowerCase: {
    message: 'Must have at least one lower-case character',
```

```
    pattern: /([a-z]+)/
  },
  special:{
    message: 'Must have at least one special character (#$@!&%...)',
    pattern: /([\!\@\#\$\%\^\&\*\(\)\_\+\{\}\:\"\<\>\?\\|\[\]\/'\,\.\`\~]+)/
  },
  number: {
    message: 'Must have at least one number',
    pattern: /([0-9]+)/
  },
  'over6': {
    message: 'Must be more than 6 characters',
    pattern: /(.{6,})/
  }
}
```

Фактически имеется набор правил, с каждым из которых связаны:

- ключ (например, `over6`);
- сообщение, например *Must be more than 6 characters*;
- регулярное выражение (например, `/(.{6,})/`).

Переходим к файлу password.jsx. Необходимо сделать следующее:

- выполнить рендер с правилами `upperCase`, `lowerCase`, `special`, `number` и `over6`;
- убедиться в том, что рендер выполнен успешно (длина=5);
- проверить, что правило 1 не выполняется;
- щелкнуть на кнопке Generate;
- проверить, что правило 2 выполняется.

В реализации компонента импортируются зависимости, а компонент создается с исходным состоянием (ch16/password/jsx/password.jsx).

Листинг 19.7. Реализация компонента Password

```
const React = require('react')
const ReactDOM = require('react-dom')
const generatePassword = require('../js/generate-password.js')

const rules = require('../js/rules.js')

const PasswordGenerate = require('./password-generate.jsx')
const PasswordInfo = require('./password-info.jsx')
const PasswordInput = require('./password-input.jsx')
const PasswordVisibility = require('./password-visibility.jsx')

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

```
    this.state = {strength: {}, password: '', visible: false, ok: false}
    this.generate = this.generate.bind(this)
    this.checkStrength = this.checkStrength.bind(this)
    this.toggleVisibility = this.toggleVisibility.bind(this)
  }
  ...
}
```

Затем реализуется метод для проверки надежности пароля:

```
checkStrength(event) {
  let password = event.target.value
  this.setState({password: password})
  let strength = {}
```

Следующий блок кода перебирает все свойства (`upperCase`, `over6` и т. д.) и проверяет текущий пароль по регулярному выражению в правиле. Если критерий выполняется, то свойству в объекте `strength` присваивается значение `true`:

```
Object.keys(this.props).forEach((key, index, list)=>{
  if (this.props[key] && rules[key].pattern.test(password)) {
    strength[key] = true
  }
})
```

Метод `this.setState()` является асинхронным, поэтому для предоставления логики, зависящей от обновленного состояния, используется обратный вызов. В данном случае мы проверяем, что количество свойств в объекте `strength` (`this.state.strength`) равно количеству правил (`props`). Это очень простая проверка; правильнее было бы проверять каждое свойство в цикле, но пока достаточно и этого кода. `ok` присваивается значение `true`, если числа совпадают (то есть если выполняются все критерии надежности пароля):

```
this.setState({strength: strength}, ()=>{
  if (Object.keys(this.state.strength).length ==
    Object.keys(this.props).length) {
    this.setState({ok: true})
  } else {
    this.setState({ok: false})
  }
})
```

Следующий метод скрывает и отображает поле пароля. Это может быть полезно при генерировании нового пароля на случай, если вы захотите сохранить пароль (или если вы забыли его):

```
toggleVisibility() {
  this.setState({visible: !this.state.visible}, ()=>{
  })
}
```

Далее идет метод `generate()`, который создает случайные пароли с использованием модуля js/generate-password.js. Присваивание `visible` значения `true` гарантирует, что пользователь увидит сгенерированный пароль. Прямо после генерирования пароля вызывается метод `checkStrength()` для проверки силы. В нормальной ситуации условия будут выполнены, а пользователь сможет продолжить работу, щелкнув на кнопке Save:

```
generate() {
  this.setState({visible: true, password: generatePassword()}, ()=>{
    this.checkStrength({target: {value: this.state.password}})
  )
}
```

В функции `render()` компонент `Password` обрабатывает правила и рендерит ряд других компонентов React:

- `PasswordInput` — поле для ввода пароля (`input`).
- `PasswordVisibility` — флаг видимости `Password` (`input` с типом `checkbox`).
- `PasswordInfo` — список правил надежности пароля (`ul`).
- `PasswordGenerate` — кнопка генерирования пароля (`button`).

Все начинается с обработки правил и определения того, какие из них выполняются (`isCompleted`). Вместо передачи контекста в `_this` или использования паттерна `bind(this)` используются «стрелочные» функции `()=>{}`. Принципиальных различий нет, выберите тот или иной вариант и используйте его.

`Object.keys` деструктурирует хеш-таблицу в массив, возвращая массив ключей заданного объекта. Вы можете перебрать массив ключей при помощи `map()` и построить новый массив с объектами, содержащими свойства `key`, `rule` и `isCompleted`:

```
render() {
  var processedRules = Object.keys(this.props).map((key)=>{
    if (this.props[key]) {
      return {
        key: key,
        rule: rules[key],
        isCompleted: this.state.strength[key] || false
      }
    }
  })
  // return ...
```

Реализация функции render()

После того как массив обработанных правил будет готов, можно переходить к рендерингу компонентов. Помните, что `for` является зарезервированным словом в JavaScript, именно поэтому необходимо использовать `className`, а не `class` (ch16/password/jsx/password.jsx).

Листинг 19.8. Реализация render()

```
return (
  <div className="well form-group col-md-6">
    <label>Password</label>
    <PasswordInput
      name="password"
      onChange={this.checkStrength}
      value={this.state.password}
      visible={this.state.visible}/>
    <PasswordVisibility
      checked={this.state.visible}
      onChange={this.toggleVisibility}/>
    <PasswordInfo rules={processedRules}/>
    <PasswordGenerate onClick={this.generate}>
      Generate
    </PasswordGenerate>
    <button className={'btn btn-primary' +
      ➥ ((this.state.ok)? '': ' disabled')}>
      Save
    </button>
  </div>
)
```

Проверяет надежность пароля при каждом изменении в поле ввода

Скрывает и отображает поле при изменении состояния флажка

Генерирует новый пароль при щелчке на кнопке Generate

Рассмотрим самые важные части более подробно. `PasswordInput` является управляемым компонентом (за подробным сравнением управляемых и неуправляемых компонентов обращайтесь к главе 5). Вы прослушиваете любые изменения с помощью обратного вызова `this.checkStrength`, который использует `e.target.value`, так что необходимости в ссылках нет:

```
<PasswordInput name="password" onChange={this.checkStrength}
➥ value={this.state.password} visible={this.state.visible}/>
```

`PasswordVisibility`, как и `PasswordInput`, является управляемым компонентом, и обработчиком события для изменений является `this.toggleVisibility`:

```
<PasswordVisibility checked={this.state.visible}
➥ onChange={this.toggleVisibility}/>
```

Объект `processedRules` передается списку правил, а кнопка `PasswordGenerate` инициирует `this.generate`:

```
<PasswordInfo rules={processedRules}/>
<PasswordGenerate onClick={this.generate}>Generate</PasswordGenerate>
```

Доступность кнопки Save определяется значением `this.state.ok`. Не забудьте пробел перед `disabled`, иначе вы получите класс `btn-primarydisabled` вместо двух классов, `btn-primary` и `disabled`:

```
<button className={'btn btn-primary' +
           ((this.state.ok)? '': ' disabled')}>Save</button>
       </div>
    )
}})
```

Другие компоненты в листингах 19.9 (ch16/password/jsx/password-generate.jsx), 19.10 (ch16/password/jsx/password-input.jsx) и 19.11 (ch16/password/jsx/password-visibility.jsx) являются презентационными. Они просто выполняют прорисовку классов и передают свойства.

Листинг 19.9. Компонент PasswordGenerate

```
const React = require('react')
class PasswordGenerate extends React.Component{
  render() {
    return (
      <button {...this.props} className="btn generate-btn">
        {this.props.children}</button>
    )
  }
}
module.exports = PasswordGenerate
```

Листинг 19.10. Компонент PasswordInput

```
const React = require('react')
class PasswordInput extends React.Component {
  render() {
    return (
      <input className="form-control"
      type={this.props.visible ? 'text' : 'password'}
      name={this.props.name}
      value={this.props.value}
      onChange={this.props.onChange}/>
    )
  }
}

module.exports = PasswordInput
```

Листинг 19.11. Компонент PasswordVisibility

```
const React = require('react')
class PasswordVisibility extends React.Component {
  render() {
    return (
      <label className="form-control">
        <input className=""
          type="checkbox"
          checked={this.props.checked}
          onChange={this.props.onChange}/> Show password
      </label>
    )
  }
}

module.exports = PasswordVisibility
```

Управляет компонентом при помощи значения свойства

Срабатывает даже для родителя через свойство

Компонент `PasswordInfo` будет рассмотрен ниже (ch16/password/jsx/password-info.jsx). Он получает массив правил `rules` и перебирает элементы этого свойства. Если значение `isCompleted` истинно, то к `<li>` добавляется тег `<strike>` (`<strike>` — тег

HTML, применяющий перечеркивание к тексту). Это условие также проверяется и в тесте password.test.js.

Листинг 19.12. Компонент PasswordInfo

```
const React = require('react')
class PasswordInfo extends React.Component {
  render() {
    return (
      <div>
        <h4>Password Strength</h4>
        <ul>
          {this.props.rules.map(function(processedRule, index, list){
            if (processedRule.isCompleted)
              return <li key={processedRule.key}>
                <strike>{processedRule.rule.message}</strike>
              </li>
            else
              return <li key={processedRule.key}>
                ➥{processedRule.rule.message}</li>
          })}
        </ul>
      </div>
    )
  }
}

module.exports = PasswordInfo
```

Проверяет выполнение правила через свойство

Использует текст, заданный в rules.js, через свойство

Работа над файлом password.jsx завершена! Теперь у вас есть все необходимое для повторного запуска теста. Не забудьте перекомпилировать приложение командой `npm run build` или `npm run build-watch`. Если вы внимательно выполнили все инструкции, после выполнения `npm test` результат должен выглядеть примерно так:

```
Using Jest CLI v0.5.10
  PASS __tests__/generate-password.test.js (0.03s)
  PASS __tests__/password.test.js (1.367s)
2 tests passed (2 total)
Run time: 2.687s
```

Поздравляю, отличная работа!

19.5. Виджет в действии

Чтобы увидеть виджет в действии, необходимо сделать еще один маленький шаг: создать jsx/app.jsx, файл примера для компонента. Вот как выполняется рендер виджета `Password` в приложении:

```
const React = require('react')
const ReactDOM = require('react-dom')
const Password = require('./password.jsx')

ReactDOM.render(<Password
```

```
    upperCase={true}
    lowerCase={true}
    special={true}
    number={true}
    over6={true}/>,
  document.getElementById('password'))
```

Файлы запускаются так же, как и в любом другом приложении клиентской части. Я предпочитаю node-static (https://github.com/cloudhead/node-static), хотя вы также можете просмотреть демоверсию по адресу http://reactquickly.co/demos. Обратите внимание на то, как кнопка Save становится активной при выполнении всех правил (рис. 19.3).

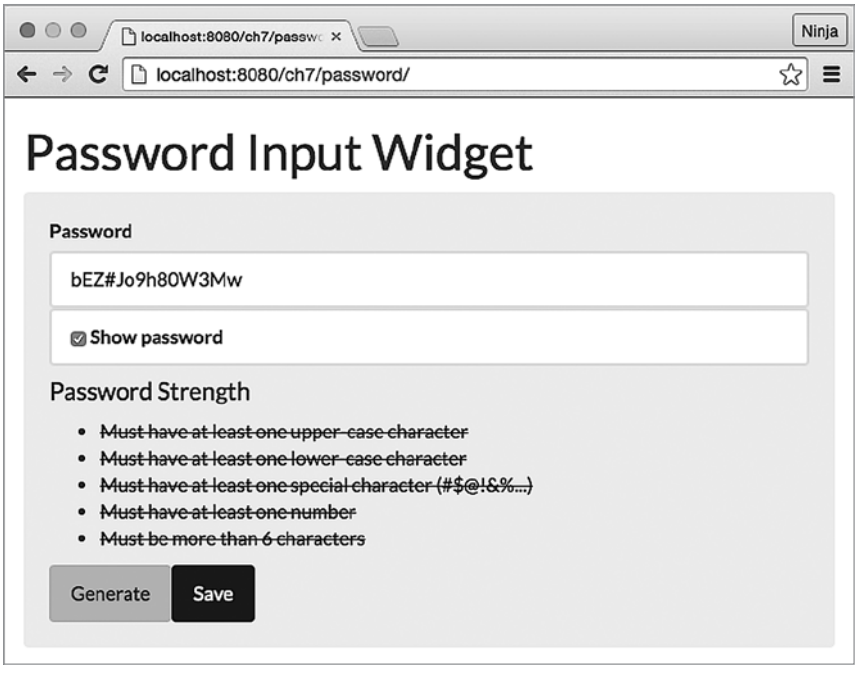


Рис. 19.3. Кнопка Save становится доступной при соблюдении всех критериев надежности

CI и CD

Самые передовые методы программирования не ограничиваются написанием и локальным запуском тестов. Тесты становятся намного более ценными, когда они объединяются с процессом развертывания и автоматизируются. Эти процессы, называемые «непрерывной интеграцией» (CI, Continuous Integration) и «непрерывным развертыванием» (CD, Continuous Deployment), отлично подходят для ускорения и автоматизации распространения ПО.

Я настоятельно рекомендую настроить CI/CD практически для любого программного продукта, за исключением разве что прототипов. Существует множество хороших решений SaaS (Software-as-a-Service) и решений с собственным сервером (self-hosted) решений. С тестами этого проекта настройка среды CI/CD не займет много времени. Например, с AWS, Travis CI или CircleCI достаточно настроить ваш проект в контексте среды, в которой он должен выполняться, а затем передать команду выполнения теста — например, npm test. Вы даже можете интегрировать эти средства SaaS CI с GitHub, чтобы вы и ваша команда видели сообщения CI (тест прошел, тест не прошел, количество и местонахождение ошибок) для pull-запросов GitHub.

Amazon Web Services предлагает собственные управляемые сервисы: CodeDeploy, CodePipeline и CodeBuild. За дополнительной информацией об этих сервисах AWS обращайтесь на сайт Node University: https://node.university/p/aws-intermediate. Если вы предпочитаете решение с собственным сервером вместо управляемого, присмотритесь к Jenkins (https://jenkins.io) и Drone (https://github.com/drone/drone).

19.6. Домашнее задание

Для получения дополнительных баллов сделайте следующее:

- Протестируйте любую ситуацию, которая придет вам в голову: например, введите только символ нижнего регистра (например, *r*) и убедитесь в том, что выполняется только критерий наличия символов нижнего регистра, но не другие критерии.
- Создайте бесплатную учетную запись в облачном провайдере SaaS CI (AWS, Travis CI, CircleCI и т. д.) и настройте проект для выполнения в облачной среде CI.

Отправьте свой код в новую папку в ch16 как pull-запрос в репозиторий GitHub этой книги: https://github.com/azat-co/react-quickly.

19.7. Итоги

- По соглашению Jest тестирует файлы, хранящиеся в папке `__tests__`.
- Вы можете использовать обычный или неглубокий рендеринг с `react-dom/test-utils` или `react-test-renderer/shallow`.
- Тесты Jest (версия 19) могут писаться на JSX, потому что Jest преобразует JSX автоматически.
- Чтобы включить автоматический перезапуск тестов (рекомендуется для разработки), используйте команду `jest --watch`.

20 Проект: реализация автозаполнения с Jest, Express и MongoDB

Посмотрите вступительный видеоролик к этой главе, отсканировав QR-код или перейдя по ссылке http://reactquickly.co/videos/ch20.

ЭТА ГЛАВА ОХВАТЫВАЕТ СЛЕДУЮЩИЕ ТЕМЫ:

- Структура проекта и конфигурация Webpack.
- Реализация веб-сервера.
- Добавление браузерного сценария.
- Создание серверного шаблона.
- Реализация компонента Autocomplete.

В этом проекте мы прежде всего постараемся объединить многие концепции, представленные в книге, — компоненты, состояния, элементы форм и тестирование, а также получение данных от сервера API и хранилища и реализацию простого сервера Express и универсальный React. Большая часть всех этих концепций уже была представлена в книге, но повторение — мать учения, особенно повторение нечастое!

В этой главе мы построим достаточно полноценный компонент и обеспечим его работу с подсистемой хранения данных. Этот маленький проект достаточно близок к реальным проектам, которые вы будете создавать на работе.

Вкратце: этот проект показывает, как строится компонент автозаполнения, по внешнему виду и функциональности близкий к компоненту Slack (популярная

система обмена сообщениям) и Google (популярная поисковая система), как показано на рис. 20.1. Для простоты этот виджет будет работать с названиями комнат в чат-приложении.

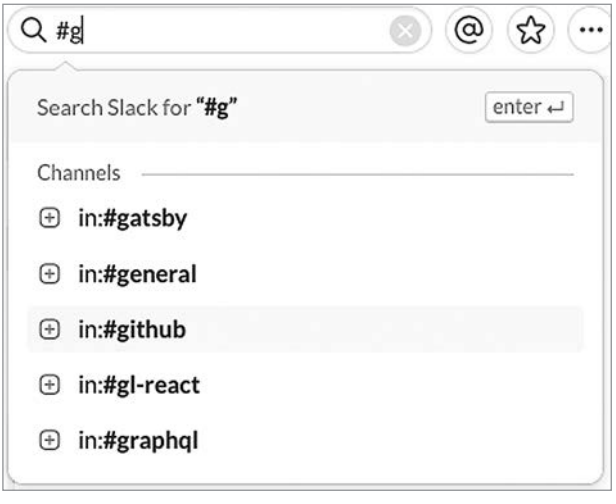


Рис. 20.1. В приложении Slack в начале ввода текста виджет предлагает варианты завершения

Виджет автозаполнения, показанный на рис. 20.2, содержит следующие составляющие.

1. *Поле ввода* — присутствует всегда, но в исходном состоянии не содержит данных.
2. *Список вариантов, отфильтрованный в соответствии с введенными символами*, — отображается при наличии хотя бы одного совпадения.
3. *Кнопка Add* — отображается при отсутствии совпадений.

Названия комнат фильтруются по совпадению введенных символов с начальными символами каждого варианта (рис. 20.3). Например, если имеются комнаты с названиями *angular*, *angular2* и *react*, то при вводе символов *angu* останутся только варианты *angular* и *angular2*, но не вариант *react*.

А если совпадений вообще нет? Тогда вы сможете добавить новый вариант при помощи кнопки Add. Для удобства кнопка Add отображается только при отсутствии совпадений (рис. 20.4). Она сохраняет новый вариант в базе данных.

Новый вариант сохраняется в базе данных вызовом XHR к REST API. Название новой комнаты может использоваться при будущем поиске (рис. 20.5) наряду с исходными названиями.

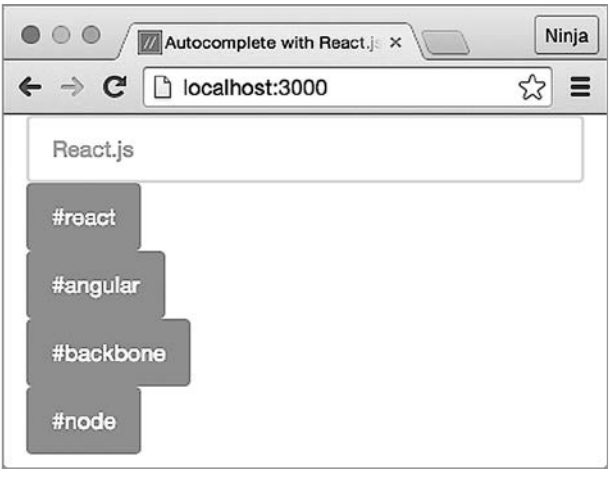


Рис. 20.2. Форма автозаполнения с пустым полем

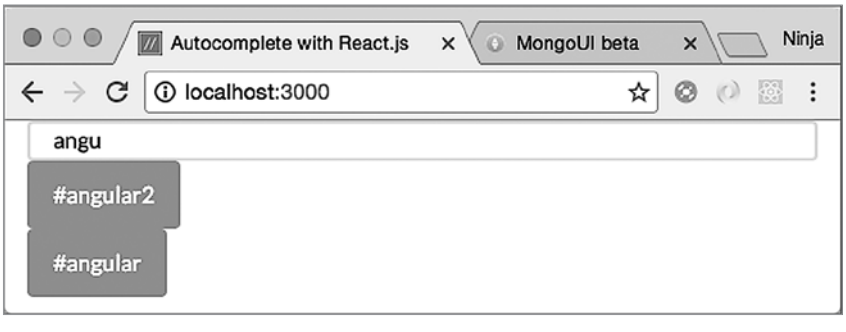


Рис. 20.3. При вводе символов angu список фильтруется, и в нем остаются только варианты angular и angular2

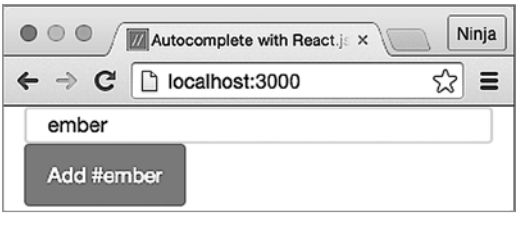


Рис. 20.4. Кнопка Add появляется только при отсутствии совпадений

Для реализации виджета автозаполнения вам понадобится:

- Установить зависимости.
- Настроить процесс сборки с использованием Webpack.

- Написать тесты с использованием Jest.
- Реализовать сервер Express REST API, который подключается к MongoDB, а также выполняет функции статического сервера для примера приложения с виджетом.
- Реализовать компонент React `Autocomplete`.
- Реализовать пример с использованием `Autocomplete` и `Handlebars`.

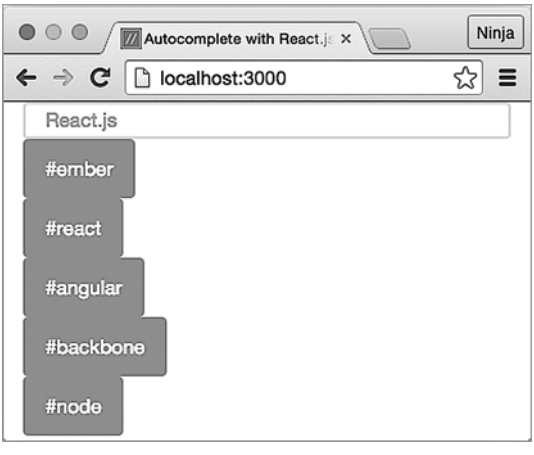


Рис. 20.5. Название комнаты было сохранено, и теперь оно присутствует в списке

Мы выполним рендер компонентов React на сервере, протестируем их с Jest, а затем сгенерируем запросы AJAX/XHR с использованием `axios`.

> **ПРИМЕЧАНИЕ** Исходный код примеров этой главы доступен по адресам www.manning.com/books/react-quickly и https://github.com/azat-co/react-quickly/tree/master/ch20. Также некоторые демонстрационные примеры доступны по адресу http://reactquickly.co/demos.

Начнем с настройки проекта.

20.1. Структура проекта и конфигурация Webpack

Чтобы вы получили представление о технологическом стеке, в этом проекте будут использоваться следующие технологии и библиотеки:

- Node.js и npm для компиляции JSX и загрузки зависимостей (например, React).
- Webpack как инструмент сборки.

- Jest как механизм тестирования.
- Express для выполнения функций веб-сервера; MongoDB для хранения вариантов автозаполнения (с обращением через драйвер MongoDB для Node.js).
- Handlebars для определения структуры.

ПОЧЕМУ НЕ ИСПОЛЬЗОВАТЬ REACT ДЛЯ ВСЕГО?

Я предпочитаю использовать Handlebars для определения структуры по нескольким причинам. Во-первых, в React выводить неэкранированную разметку HTML достаточно трудно; для этого используется странный и небезопасный синтаксис. С другой стороны, именно это нужно для универсального React и рендеринга на стороне сервера. Да, неэкранированная разметка HTML может создать угрозу межсайтовых сценарных атак[1], но при рендере разметки на сервере строка HTML находится под вашим контролем.

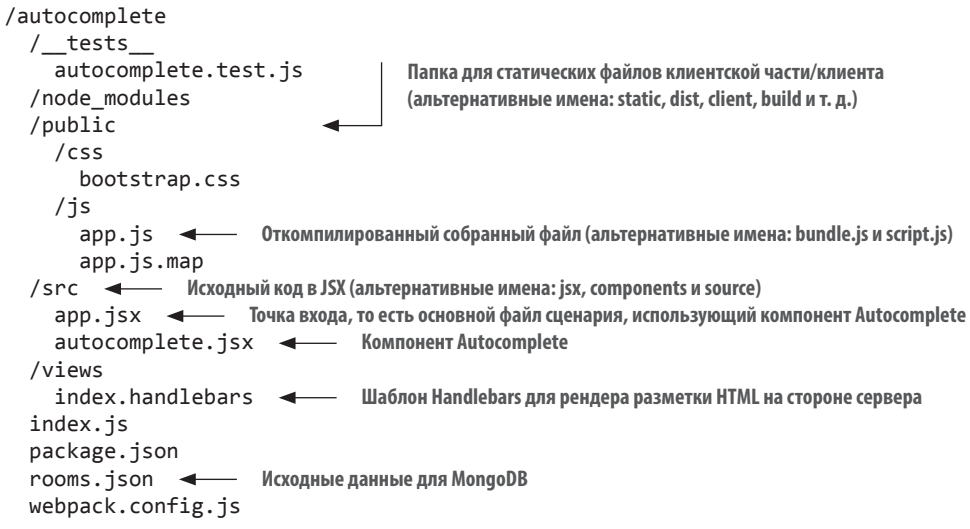
Во-вторых, Handlebars более естественно рендерит такие конструкции, как <!DOCTYPE html>. React не может делать это так же естественно, потому что React в большей степени ориентируется на отдельные элементы, а не на целые страницы.

В-третьих, React специализируется на управлении состоянием и автоматической синхронизации представления с состоянием. Если вы ограничиваетесь рендерингом статических строк HTML из компонентов React, то зачем вообще использовать React? Это перебор. Handlebars имеет много общего с HTML, поэтому вы можете легко копировать существующий код HTML, не задумываясь о JSX и ловушках React, которые могут подстерегать вас при преобразовании HTML в React.

Наконец, мой личный опыт объяснения функциональности кода другим разработчикам и учащимся на курсах и семинарах показал, что некоторым людям труднее понять структуру приложения, когда одни компоненты React используются для определения структуры на сервере, а другие — для представлений как в клиенте, так и на сервере.

Установка всех необходимых программ рассматривается в приложении А, и я не стану утомлять вас повторениями. Создайте папку нового проекта с именем autocomplete. Структура папок выглядит так:

[1] При межсайтовых сценарных атаках (XSS) атакующий внедряет вредоносный код на доверенные сайты, содержащие уязвимости XSS. Например, атакующий может отправить сообщение с вредоносным кодом, включающим элементы <script>, на уязвимый форум, который не проводит защитную обработку и/или не экранирует текст. В итоге посетители форума будут выполнять вредоносный код. За дополнительной информацией о XSS обращайтесь к книге Джейкоба Каллина (Jacob Kallin) и Ирэн Лобо Валбуэна (Irene Lobo Valbuena) «Excess XSS: A Comprehensive Tutorial on Cross-Site Scripting», https://excess-xss.com.



В папке __tests__ хранятся тесты Jest. Как вы уже знаете, папка node_modules предназначена для зависимостей Node.js (из файла package.json для `npm`). Папки public, public/css и public/js содержат статические файлы приложения.

ОБ ИМЕНАХ

Имена играют исключительно важную роль в качественном программировании, потому что хорошее имя передает полезную информацию. По имени можно многое сказать о сценарии, файле, модуле или компоненте без чтения исходного кода, тестов или документации (которой может и не быть!).

Когда вы уже привыкли размещать файлы JSX в папке jsx и использовать build как приемную папку для скомпилированных файлов, я начал использовать другие имена. Дело в том, что на практике вы встретите много разных соглашений об именах. Вероятно, в каждом проекте будет использоваться своя структура; эти структуры могут различаться — в мелочах или значительно. Вы как разработчик должны уверенно настраивать такие инструменты, как Webpack и библиотеки (например, Express), для работы с любыми именами. По этой причине (а также ради разнообразия) в этой главе я использую имя public вместо build (к тому же имя public по соглашению используется для статических файлов, предоставляемых express), src — вместо jsx (у вас могут быть и другие файлы с исходным кодом, не только JSX, верно?) и т. д.

Файл public/js/app.js строится Webpack из зависимостей и исходного кода JSX src/app.jsx. Исходный код компонента `Autocomplete` находится в файле src/autocomplete.jsx.

Папка views предназначена для шаблонов Handlebars. Если вы уверены в своих навыках React, вам не обязательно использовать шаблонизатор; вы можете использовать React как шаблонизатор Node.js!

В корневой папке проекта размещаются следующие файлы:

- webpack.config.js — обеспечивает выполнение задач построения.
- package.json — содержит метаданные проекта.
- rooms.json — содержит исходные данные MongoDB.
- index.js — с сервером Express.js и его маршрутами для сервера API (GET и POST /rooms).

Напомню: чтобы вам не пришлось устанавливать каждую зависимость с точной версией вручную, вы можете скопировать файл package.json из листинга 20.1 (ch20/autocomplete/package.json) в корневую папку и выполнить команду `npm install`.

Листинг 20.1. Зависимости и настройка проекта

```
{
  "name": "autocomplete",
  "version": "1.0.0",
  "description": "React.js autocomplete component with Express.js, and
  ➥ MongoDB example.",
  "main": "index.js",
  "scripts": {
    "test": "jest",
    "start": "npm run build && ./node_modules/.bin/node-dev index.js",
    "build": "./node_modules/.bin/webpack",
    "seed": "mongoimport rooms.json --jsonArray --collection=rooms
    ➥ --db=autocomplete"
  },
  "keywords": [
    "react.js",
    "express.js",
    "mongodb"
  ],
  "author": "Azat Mardan",
  "license": "MIT",
  "babel": {
    "presets": [
      "react"
    ]
  },
  "dependencies": {
    "babel-register": "6.11.6",
    "body-parser": "1.13.2",
    "compression": "1.5.1",
    "errorhandler": "1.4.1",
    "express": "4.13.1",
    "express-handlebars": "2.0.1",
    "express-validator": "2.13.0",
    "mongodb": "2.0.36",
    "morgan": "1.6.1"
  },
```

Позволяет импортировать и транспилировать JSX на стороне сервера

Веб-фреймворк Express на стороне сервера

Плагин Express (промежуточный модуль) для регистрации запросов HTTP

Библиотека для подключения к базе данных MongoDB

```
  "devDependencies": {
    "axios": "0.13.1",
    "babel-core": "6.10.4",
    "babel-loader": "6.2.4",
    "babel-preset-react": "6.5.0",
    "jest-cli": "13.2.3",
    "node-dev": "3.1.3",
    "react": "15.5.4",
    "react-dom": "15.5.4",
    "webpack": "1.13.1"
  }
}
```

Конечно, если вы хотите в итоге получить работоспособное приложение, важно использовать версии, приведенные в книге. Также не забудьте установить зависимости из package.json командой `npm i`.

Раздел `scripts` выглядит интересно:

```
"scripts": {
    "test": "jest",
    "start": "./node_modules/.bin/node-dev index.js",
    "build": "./node_modules/.bin/webpack",
    "seed": "mongoimport rooms.json --jsonArray --collection=rooms
    ➥ --db=autocomplete"
  },
```

`test` предназначен для выполнения тестов Jest, а `start` — для построения и запуска вашего сервера. Также добавляются исходные данные для названий комнат, этот сценарий запускается командой `$ npm run seed`. Базе данных присвоено имя `autocomplete`, а коллекции — имя `rooms`. Содержимое файла rooms.json выглядит так:

```
[ {"name": "react"},
  {"name": "node"},
  {"name": "angular"},
  {"name": "backbone"}]
```

Результат выполнения команды seed выглядит примерно так (база данных MongoDB должна работать в отдельном процессе):

```
> autocomplete@1.0.0 seed /Users/azat/Documents/Code/
➥ react-quickly/ch20/autocomplete
> mongoimport rooms.json --jsonArray --collection=rooms --db=autocomplete

2027-07-10T07:06:28.441-0700 connected to: localhost
2027-07-10T07:06:28.443-0700 imported 4 documents
```

Вы определили зависимости проекта; теперь нужно настроить процесс сборки Webpack, чтобы вы могли использовать ES6 и выполнять преобразование JSX. Для этого создайте в корневом каталоге файл webpack.config.js, содержащий код из листинга 20.2 (ch20/autocomplete/webpack.config.js).

Листинг 20.2. Конфигурация Webpack

```
module.exports = {
  entry: './src/app.jsx',
  output: {
    path: __dirname + '/public/js/',
    filename: 'app.js'
  },
  devtool: '#sourcemap',
  stats: {
    colors: true,
    reasons: true
  },
  module: {
    loaders: [
      {
        test: /\.jsx?$/,
        exclude: /(node_modules)/,
        loader: 'babel-loader'
      }
    ]
  }
}
```

Назначает точку входа для проекта (таких точек может быть несколько)

Настраивает карты исходного кода для просмотра правильных номеров строк в DevTools

Применяет Babel с использованием конфигураций Babel из package.json

Этот конфигурационный файл Webpack не отличается от файлов, которые использовались в других проектах, построенных нами ранее. Он настраивает Babel для транспиляции файлов JSX и определения папки, в которой должен сохраняться упакованный файл JavaScript.

20.2. Реализация веб-сервера

В этом проекте вместо основного файла HTML нужно написать простой веб-сервер, который будет получать запросы на основании данных, введенных пользователем к настоящему моменту, и отвечать списком вариантов. Он также будет рендерить элемент управления на стороне сервера и передавать соответствующую разметку HTML клиенту. Как упоминалось ранее, в примерах используется веб-сервер Express. Файл index.js определяет веб-сервер и состоит из трех разделов:

- Импортирование библиотек и компонентов.
- Определение REST API для входных запросов.
- Рендер элемента управления на стороне сервера.

Рассмотрим все три раздела поочередно. Первая часть — импортирование — самая тривиальная. В листинге 20.3 представлены все компоненты и библиотеки, необходимые серверу (ch20/autocomplete/index.js).

Листинг 20.3. Компоненты и библиотеки для веб-сервера

```
const express = require('express'),
  mongodb = require('mongodb'),
  app = express(),
  bodyParser = require('body-parser'),
  validator = require('express-validator'),
  logger = require('morgan'),
  errorHandler = require('errorhandler'),
  compression = require('compression'),
  exphbs = require('express-handlebars'),
  url = 'mongodb://localhost:27017/autocomplete',
  ReactDOM = require('react-dom'),
  ReactDOMServer = require('react-dom/server'),
  React = require('react')

require('babel-register')({
  presets: ['react']
})
const Autocomplete = ,
React.createFactory(require('./src/autocomplete.jsx')),
  port = 3000
...
```

Определяет и импортирует многострочное объявление (как если бы ключевое слово const присутствовало в каждой строке)

Создает экземпляр приложения Express

Определяет строку подключения к локальной базе данных MongoDB

Определяет конфигурацию babel-register для импортирования файлов JSX

Создает функцию-фабрику компонента React на основе файла JSX (будет возвращать новые экземпляры; использовать createElement() не нужно)

В следующем разделе мы продолжим рассмотрение файла index.js и обсудим ту его часть, которая связана с подключением к базе данных и с промежуточными модулями.

20.2.1. Определение REST-совместимых API

Файл index.js содержит GET- и POST-маршруты для `/rooms`. Они предоставляют REST-совместимые конечные точки API для получения данных вашим клиентским приложением. В свою очередь, данные берутся из базы данных MongoDB, на что указывает сценарий npm (`npm run seed`); предполагается, что этот сценарий присутствует в файле package.json, а в проекте присутствует файл rooms.json. Но прежде чем загружать данные из базы данных, необходимо подключиться к ней и определить маршруты Express (ch20/autocomplete/index.js).

Листинг 20.4. Маршруты REST-совместимого API

```
mongodb.MongoClient.connect(url, function(err, db) {
  if (err) {
    console.error(err)
    process.exit(1)
  }
  app.use(compression())
  app.use(logger('dev'))
  app.use(errorHandler())
  app.use(bodyParser.urlencoded({extended: true}))
```

Подключается к MongoDB

Завершает текущий процесс с кодом ошибки

```
app.use(bodyParser.json())
app.use(validator())
app.use(express.static('public'))
app.engine('handlebars', exphbs())
app.set('view engine', 'handlebars')

app.use(function(req, res, next){
  req.rooms = db.collection('rooms')
  return next()
})

app.get('/rooms', function(req, res, next) {
  req.rooms
    .find({}, {sort: {_id: -1}})
    .toArray(function(err, docs) {
      if (err) return next(err)
      return res.json(docs)
    }
  )
})

app.post('/rooms', function(req, res, next) {
  req.checkBody('name', 'Invalid name in body')
    .notEmpty()
  var errors = req.validationErrors()
  if (errors) return next(errors)
  req.rooms.insert(req.body, function (err, result) {
    if (err) return next(err)
    return res.json(result.ops[0])
  })
})
```

Возвращает список существующих комнат в чате

Создает новую комнату в чате

Проверяет, что в данных присутствует имя и тело не пусто

Обращается с вызовом к базе данных для сохранения нового сообщения

Если вам понадобится освежить в памяти API Express.js, в приложении B имеется удобная сводка.

20.2.2. Рендеринг React на сервере

Наконец, файл index.js содержит маршрут /, по которому React на стороне сервера рендерит разметку, объединяя компоненты с объектами комнат (ch20/autocomplete/index.js).

Листинг 20.5. React на стороне сервера

```
app.get('/', function(req, res, next){
    var url = 'http://localhost:3000/rooms'
    req.rooms.find({}, {sort: {_id: -1}}).toArray(function(err, rooms){
      if (err) return next(err)
      res.render('index', {
        autocomplete: ReactDOMServer.renderToString(Autocomplete({
```

Создает элемент React Autocomplete

```
        options: rooms,   <-- Передает названия комнат в свойстве options
        url: url          <-- Передает URL-адрес API для получения и создания имен
      })),
      data: `<script type="text/javascript">
              window.__autocomplete_data = {
                rooms: ${JSON.stringify(rooms, null, 2)},   <-- Использует параметры stringify для украшения вывода
                url: "${url}"
              }
            </script>`
    })
  })
})
```

Передает данные от сервера браузерному коду, чтобы обеспечить правильность работы универсального React

Компонент `Autocomplete` содержит два свойства: `options` и `url`. В свойстве `options` хранятся названия комнат чата, а в свойстве `url` — URL сервера API (в данном случае http://localhost:3000/rooms). Компонент `Autocomplete` также будет рендериться в браузере.

20.3. Добавление браузерного сценария

Браузерный сценарий — пример возможного использования виджета автозаполнения, он будет выполняться только в браузере. Файл получается очень коротким — он просто создает элемент со свойствами `options` и `url` (ch20/autocomplete/src/app.jsx).

Листинг 20.6. Основной сценарий на стороне клиента

```
const React = require('react')
const ReactDOM = require('react-dom')

const Autocomplete = require('./autocomplete.jsx')
const {rooms, url} = window.__autocomplete_data   <-- Получает данные из глобальной переменной

ReactDOM.render(<Autocomplete                     <-- Создает и рендерит компонент с использованием существующих данных и без запросов XHR
    options={rooms}
    url={url}/>,
  document.getElementById('autocomplete')
)
```

Глобальное значение `__autocomplete_data` передается через данные шаблона `data` с использованием тега `<script>` в маршруте `/`.

Листинг 20.7. Приложение Express рендерит данные для React в браузере

```
res.render('index', {
  // ...
  data: `<script type="text/javascript">   <-- Использует элемент script для «вывода» JavaScript в шаблоне Handlebars index.hbs
          window.__autocomplete_data = {
```

```
        rooms: ${JSON.stringify(rooms, null, 2)},
        url: "${url}"
      }
    </script>`
```

Преобразует данные из объекта в строку для вывода

Тег HTML `<script>` внедряется в шаблон index.hbs (Express предполагает расширение файла .hbs, поэтому его указывать не обязательно). На следующем шаге мы займемся реализацией этого шаблона.

20.4. Создание серверного шаблона

В файле index.hbs выводятся значения `props` и `autocomplete`.

Листинг 20.8. Основная страница разметки

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <title>Autocomplete with React.js</title>
    <meta name="description" content="React Quickly: Autocomplete" />
    <meta name="author" content="Azat Mardan" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <link type="text/css" rel="stylesheet" href="/css/bootstrap.css" />
  </head>

  <body>
  <div class="container-fluid">
      <div>{{{data}}}</div>
      <div class="row-fluid">
        <div class="span12">
          <div id="content">
            <div class="row-fluid"
              id="autocomplete" />{{{autocomplete}}}</div>
        </div>
      </div>
    </div>
    <script type="text/javascript" src="/js/app.js"></script>
  </body>
</html>
```

Рендерит тег <script> со списком названий и URL для API

Рендерит статическую разметку HTML с контрольной суммой универсального React

Применяет клиентский сценарий, который активизирует React в браузере и использует __autocomplete_data (см. предыдущий раздел)

Подготовка к запуску примера завершена. Разумеется, центральное место в нем занимает компонент `Autocomplete`. На следующем шаге мы наконец-то займемся его реализацией.

20.5. Реализация компонента Autocomplete

Компонент `Autocomplete` автономен; это означает, что он не только является компонентом представления, но и может выполнять чтение и сохранение данных через REST API. Он обладает двумя свойствами: `options` и `url`. В соответствии с канонами TDD начнем программирование компонента `Autocomplete` с тестов.

20.5.1. Тесты для Autocomplete

Принципы TDD/BDD требуют начинать с тестов. В файле __tests__/autocomplete.test.js перечислены названия комнат, после чего компонент рендерится в `autocomplete`:

```
const rooms = [   ← Фиксированные данные с названиями комнат
    { "_id" : "5622eb1f105807ceb6ad868b", "name" : "node" },
    { "_id" : "5622eb1f105807ceb6ad868c", "name" : "react" },
    { "_id" : "5622eb1f105807ceb6ad868d", "name" : "backbone" },
    { "_id" : "5622eb1f105807ceb6ad868e", "name" : "angular" }
  ]

const TestUtils = require('react-addons-test-utils'),
  React = require('react'),
  ReactDOM = require('react-dom'),
  Autocomplete = require('../src/autocomplete.jsx'),
  fD = ReactDOM.findDOMNode                    ← Сохраняет объект fD для удобства (меньше вводимых символов — меньше ошибок)

const autocomplete = TestUtils.renderIntoDocument(   ← Использует TestUtils из react-addons-test-utils для рендера компонента Autocomplete
  React.createElement(Autocomplete, {
    options: rooms,
    url: 'test'
  })
)
const optionName = TestUtils.findRenderedDOMComponentWithClass(autocomplete,
➥ 'option-name')   ← Получает поле ввода по классу option-name
...
```

Программа получает поле ввода, которому назначен класс `option-name`. Среди вариантов названий комнат компонент будет искать совпадения для текущего значения поля ввода.

Теперь можно переходить к написанию тестов. Вы можете получить все элементы `option-name` из виджета и сравнить их количество с 4 (количество комнат в массиве `rooms`):

```
describe('Autocomplete', () => {
  it('have four initial options', () => {
    var options = TestUtils.scryRenderedDOMComponentsWithClass(
      autocomplete,
      'option-list-item'
    )
    expect(options.length).toBe(4)
  })
```

Следующий тест изменяет значение поля ввода, после чего проверяет это значение и количество предлагаемых вариантов автозаполнения. В данном случае совпадение должно быть только одно, а именно `react`:

```
it('change options based on the input', () => {
    expect(fD(optionName).value).toBe('')
    fD(optionName).value = 'r'
    TestUtils.Simulate.change(fD(optionName))
    expect(fD(optionName).value).toBe('r')
    options = TestUtils.scryRenderedDOMComponentsWithClass(autocomplete,
      'option-list-item')
    expect(options.length).toBe(1)
    expect(fD(options[0]).textContent).toBe('#react')
  })
```

В последнем тесте название комнаты заменяется на `ember`. Совпадений быть не должно, только кнопка Add:

```
it('offer to save option when there are no matches', () => {
    fD(optionName).value = 'ember'
    TestUtils.Simulate.change(fD(optionName))
    options = TestUtils.scryRenderedDOMComponentsWithClass(
      autocomplete,
      'option-list-item'
    )
    expect(options.length).toBe(0)
    var optionAdd = TestUtils.findRenderedDOMComponentWithClass(
      autocomplete,
      'option-add'
    )
    expect(fD(optionAdd).textContent).toBe('Add #ember')
  })
})
```

20.5.2. Код компонента Autocomplete

Наконец, можно написать компонент `Autocomplete` (ch20/autocomplete/src/autocomplete.jsx). Он включает поле ввода, список подходящих вариантов и кнопку Add для добавления нового варианта при отсутствии совпадений. Компонент выдает два вызова AJAX/XHR: для получения списка вариантов и для создания нового варианта.

Также компонент содержит два метода:

- `filter()` — выполняется для каждого нового ввода в поле `<input>`. Получает текущий введенный текст и список вариантов и присваивает состоянию новый список, который состоит только из вариантов, соответствующих текущему вводу.
- `addOption()` — выполняется при щелчке или нажатии клавиши Enter для кнопки Add. Получает значение и отправляет его серверу.

На верхнем уровне компонент `Autocomplete` выглядит так:

```
const React = require('react'),
  ReactDOM = require('react-dom'),
  request = require('axios')

class Autocomplete extends React.Component {
  constructor(props) {
    ...
  }
  componentDidMount() {
    ...
  }
  filter(event) {
    ...
  }
  addOption(event) {
    ...
  }
  render() {
    return (
      <div ...>
        <input ... onChange={this.filter}>
        </input>
        {this.state.filteredOptions.map(function(option,
        ➥ index, list) {
        ...
        })}
        ...
            <a ...onClick={this.addOption}>
              Add #{this.state.currentOption}
            </a>
        ...
      </div>
    )
  }
}

module.exports = Autocomplete
```

Получает список вариантов от сервера

Фильтрует список и оставляет только те варианты, которые соответствуют введенному тексту

Добавляет в базу новый вариант вызовом XHR к серверу

Сохраняет значение варианта, отслеживая событие браузера

Выводит список подходящих (отфильтрованных) вариантов

Вызывает метод add при щелчке на кнопке (ссылке)

Рассмотрим файл с самого начала. Импортируйте библиотеки в стиле CommonJS/Node.js; благодаря Webpack код упакован для использования браузером. Синоним `fD` определяется для удобства:

```
const React = require('react'),
  ReactDOM = require('react-dom'),
  request = require('axios')

const fD = ReactDOM.findDOMNode
```

`constructor` задает состояние и выполняет связывание. Значение `options` задается из свойств. `filteredOptions` изначально содержит те же данные, что и `options`,

а текущий вариант (содержимое поля ввода) пуст. По мере ввода символов `filteredOptions` будет становиться все короче и короче (в соответствии с введенными буквами).

В `componentDidMount()` запрос GET выполняется с использованием библиотеки `axios` (переменная `request`). Происходит примерно то же, что и с методом jQuery `$.get()`, но с обещаниями:

```
class Autocomplete extends React.Component {
  constructor(props) {
    super(props)
    this.state = {options: this.props.options,
      filteredOptions: this.props.options,
      currentOption: ''
    }
    this.filter = this.filter.bind(this)
    this.addOption = this.addOption.bind(this)
  }
  componentDidMount() {
    if (this.props.url == 'test') return true  <-- Блокирует получение данных для теста
    request({url: this.props.url})
      .then(response=>response.data)
      .then(body => {
        if(!body){
          return console.error('Failed to load')
        }
      this.setState({options: body})  <-- Присваивает результат options
      })
      .catch(console.error)
  }
  ...
```

Метод `filter()` вызывается для каждого изменения поля `<input>`. Он должен оставить только те варианты, которые соответствуют введенному тексту:

```
...
filter(event) {
  this.setState({
    currentOption: event.target.value,
    filteredOptions:
      (this.state.options.filter((option, index, list) => {  <-- Использует filter() для массива
      return (event.target.value === option.name.substr(0,  <-- Отсекает #
        event.target.value.length))
    }))
  })
}
```

Метод `addOption()` добавляет новый вариант в том случае, если ни одного совпадения не найдено; для этого он вызывает действие хранилища:

```
addOption(event) {
    let currentOption = this.state.currentOption
    request                                          Использует axios для выдачи запроса POST
      .post(this.props.url, {name: currentOption})
      .then(response => response.data)
      .then((body) => {
        if(!body){
          return console.error('Failed to save')
        }
        this.setState({
            options: [body].concat(this.state.options)
          },
          () => {
            this.filter({target: {value: currentOption}})
          }
        )
      })
      .catch(error=>{return console.error('Failed to save')})
  }
```

Использует Array.concat() для создания нового массива вместо Array.push(), потому что прямая модификация состояния считается нежелательной

Вызывает метод filter() в обратном вызове setState(), чтобы новое значение уже было сохранено в состоянии при выполнении filter()

Наконец, метод `render()` содержит управляемый компонент `<input>` со слушателем события `onChange`, `this.filter`:

```
...
render() {
  return (
    <div className="form-group">
      <input type="text"
        onKeyUp={(event) => (event.keyCode==13) ? this.addOption() : ''}
        className="form-control option-name"
        onChange={this.filter}
        value={this.currentOption}
        placeholder="React.js">
      </input>
```

`onKeyUp` может быть записан в виде метода (не обязательно анонимной встроенной функции) прямо в `{}`.

Список отфильтрованных вариантов хранится в состоянии `filteredOptions`, которое обновляется методом `filter()`. Мы перебираем его содержимое и выводим `_id` как ключи и ссылки с `option.name`:

```
  {this.state.filteredOptions.map(function(option, index, list){
    return <div key={option._id}>
      <a className="btn btn-default option-list-item"
```

Использует метод map() для вывода списка отфильтрованных вариантов

```
        href={'/#/'+option.name} target="_blank">
        #{option.name}
      </a>
    </div>
  })}
...
```

Выводит имя варианта с # как в Slack

Использует URL как значение якорного тега для каждого варианта

Последний элемент — кнопка Add — появляется только при отсутствии элементов в `filterOptions` (совпадения не обнаружены):

```
        ...
        {(()=>{
          if (this.state.filteredOptions.length == 0 &&
            this.state.currentOption!='')
            return <a className="btn btn-info option-add"
              onClick={this.addOption}>
              Add #{this.state.currentOption}
            </a>
        })()}
      </div>
    )
  }
}
```

Скрывает кнопку при наличии подходящих вариантов

Предлагает добавить текущее вводимое значение в список вариантов

Использует addOption как обработчик события onClick

Мы используем синтаксис CommonJS, так что объявление компонента `Autocomplete` и его экспортирование могут быть выполнены следующим образом:

```
module.exports = Autocomplete
```

Все готово. Отличная работа!

20.6. Все вместе

Если вы повторяли все описанные действия, то сможете установить зависимости следующей командой (если это не было сделано ранее):

```
$ npm install
```

Затем запустите приложение (до этого вы должны были запустить MongoDB командой `$ mongod`):

```
$ npm start
```

Тесты должны пройти при выполнении следующей команды:

```
$ npm test
```

Также имеется команда `npm run build` без отслеживания изменений (вам придется вручную выполнять ее при внесении изменений). Команда `npm start` выполняет `npm run build` за вас.

Также можно заполнить базу данных командой `$ npm run seed`. При этом MongoDB заполняется именами из файла ch20/autocomplete/rooms.json:

```
[ {"name": "react"},
{"name": "node"},
{"name": "angular"},
{"name": "backbone"}]
```

Компонент `Autocomplete` готов. Запустите проект — постройте его командой `npm run build` и откройте адрес http://localhost:3000 (предполагается, что MongoDB выполняется в отдельном терминальном окне). Хотя 127.0.0.1 является синонимом, вы должны использовать тот же домен, как в браузере, чтобы избежать проблем CORS/Access-Control-Allow-Origin, потому что JavaScript будет обращаться с вызовами к серверу *localhost*.

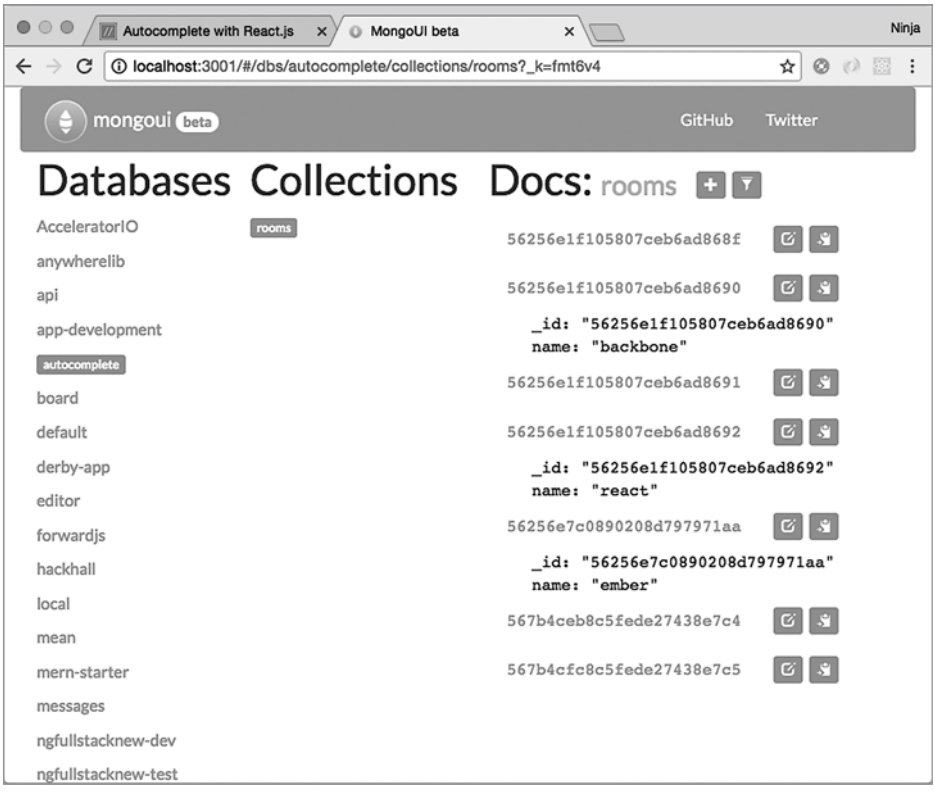
На странице должен отображаться компонент с именами (если база данных была заполнена). При вводе символов в поле ввода список фильтруется в соответствии с введенным текстом. При отсутствии совпадений щелкните на кнопке Add, чтобы добавить комнату в базу данных, — она немедленно появится в списке.

MONGO И MONGOUI

Если вам когда-либо приходилось работать с данными в MongoDB напрямую, оболочка mongo (или REPL) запускается командой mongo в терминале. Она автоматически подключается к локальному экземпляру на порте 27017 (такой экземпляр должен быть запущен заранее; используйте `mongod`). В оболочке mongo можно выполнять самые разнообразные операции: создавать документы, выдавать запросы к коллекциям, удалять базы данных и т. д. Здесь удобно то, что оболочку mongo можно использовать где угодно, даже на удаленном сервере без графического интерфейса.

Однако при работе с оболочкой mongo приходится вводить много символов, а набор с клавиатуры происходит медленно и сопряжен с высоким риском ошибок. Поэтому я построил более удобный инструмент MongoUI (https://github.com/azat-co/mongoui), который может использоваться для запроса, редактирования, добавления документов, удаления документов и выполнения других операций в браузере вместо копирования больших объемов JSON (MongoDB базируется на JSON и JavaScript).

MongoUI позволяет работать с MongoDB через удобный веб-интерфейс. На иллюстрации показаны названия комнат из моей коллекции rooms в базе данных autocomplete.



Веб-интерфейс MongoDB

Установите MongoUI командой `npm i -g mongoui`, запустите его командой mongoui, а затем откройте в браузере адрес http://localhost:3001. Кстати, приложение MongoUI построено с использованием React, Express и Webpack. Пользуйтесь!

Результат выполнения приложения показан на рис. 20.6. Вы можете открыть вкладку Network и щелкнуть на узле Localhost, чтобы убедиться в том, что рендеринг на стороне сервера работает (то есть данные и HTML для названий находятся на своих местах).

Если по какой-то причине ваш проект не работает, возможно, вышла новая версия или в коде была допущена опечатка. Работоспособный код доступен по адресу www.manning.com/books/react-quickly или https://github.com/azat-co/react-quickly/tree/master/ch20.

Рис. 20.6. Чтобы проанализировать ответ localhost, щелкните на вкладке Network (1) и узле Localhost (2) и убедитесь в том, что рендеринг разметки на стороне сервера (3) работает правильно

20.7. Домашнее задание

Для получения дополнительных баллов сделайте следующее:

- Добавьте тест для кнопки Remove (кнопка со значком X рядом с каждым вариантом).
- Добавьте кнопку Remove со значком X рядом с каждым вариантом. Реализуйте вызов AJAX/XHR и добавьте конечную точку REST для реализации удаления.

- Доработайте алгоритм поиска совпадений, чтобы он находил совпадения в середине имен. Например, при вводе символов `ac` должны выводиться варианты `react` и `backbone`, потому что в обоих вариантах встречаются буквы `ac`.
- Добавьте хранилище Redux.
- Реализуйте GraphQL вместо REST API в серверной части.

Отправьте свой код в новую папку в `ch20` как pull-запрос в репозиторий GitHub этой книги: https://github.com/azat-co/react-quickly.

20.8. Итоги

- Фигурные скобки выводят неэкранированную разметку HTML в Handlebars, тогда как в React необходимо использовать `__html` для небезопасного задания внутренней разметки HTML.
- `findRenderedDOMComponentWithClass()` пытается найти один компонент по имени его класса CSS, а `scryRenderedDOMComponentsWithClass()` находит несколько компонентов по имени класса CSS (см. главу 16).
- `babel-register` позволяет импортировать и использовать файлы JSX: `require('babel-register')({presets:[ 'react' ]})`.
- MongoUI — веб-интерфейс с открытым кодом на базе React для разработки и администрирования баз данных MongoDB. Интерфейс MongoUI устанавливается командой `npm i -g mongoui` и запускается командой `mongoui`.

Приложения

A Установка приложений

В этом приложении приводятся инструкции по установке для следующих приложений (по состоянию на май 2017 г.):

- React v15
- Node.js v6 и npm v3
- Express v4
- Twitter Bootstrap v3
- Browserify
- MongoDB
- Babel

Установка React

Вы можете загрузить React множеством разных способов:

- Используйте ссылку на файл в сети доставки контента (CDN), например Cloudflare: https://cdnjs.cloudflare.com/ajax/libs/react/15.5.4/react.js или https://cdnjs.cloudflare.com/ajax/libs/react/15.5.4/react-dom.js (полный список: https://cdnjs.com/libraries/react).
- Загрузите файл с веб-сайта React, например http://facebook.github.io/react/downloads.html или https://github.com/facebook/react.
- Используйте npm (см. следующий раздел), например `npm install react@15 react-dom@15`. Пока вам не нужно беспокоиться о выполнении React на сервере. react.js находится в папке node_modules/react/dist.
- Используйте Bower (http://bower.io) с `bower install --save react`.
- Используйте Webpack/Grunt/Browserify/Gulp для сборки из npm модулей.

Установка Node.js

Если вы не уверены в том, установлены ли Node.js и npm в вашей системе, или не знаете, какая версия у вас установлена, выполните следующие команды в терминале /iTerm/bash/zsh/ командной строки:

```
$ node -v
$ npm -v
```

В большинстве случаев npm устанавливается с Node.js, поэтому для установки npm выполните инструкции для Node.js. Самый простой способ установить Node и npm — перейти на веб-сайт и выбрать подходящую архитектуру для вашего компьютера (Windows, macOS и т. д.): https://nodejs.org/en/download.

Пользователям macOS, у которых уже установлен Ruby (как это обычно бывает на компьютерах Mac), я настоятельно рекомендую установить Homebrew. Я сам использую этот пакетный менеджер, потому что он позволяет мне установить другие инструменты разработки, такие как базы данных и серверы. Чтобы установить Homebrew на Mac, выполните в терминале следующий код Ruby (обещаю — это последний раз, когда мы используем Ruby в этой книге!):

```
$ ruby -e "$(curl -fsSL
➥ https://raw.githubusercontent.com/Homebrew/install/master/install)"
```

После установки Homebrew обновите реестр и установите Node.js вместе с `npm`. Как я уже упоминал, npm идет вместе с Node.js, поэтому дополнительные команды не понадобятся:

```
$ brew update
$ brew install node
```

Другой замечательный инструмент, который позволяет легко переключаться между версиями Node, — Node Version Manager (nvm, https://github.com/creationix/nvm):

```
$ curl -o- https://raw.githubusercontent.com/creationix/nvm/v0.32.1/install.sh
➥ | bash
$ nvm install node
```

Вот и все! Теперь вы сможете посмотреть версии Node и npm. Если вы захотите обновить npm, используйте команду npm:

```
$ npm i -g npm@latest
```

Чтобы обновить Node, используйте `nvm` или аналогичный инструмент (`nave` или `n`). Например, в `nvm` следующая команда также переустановит новые версии пакетов:

```
$ nvm install node --reinstall-packages-from=node
```

Если npm выдает ошибки разрешений при установке модуля/пакета, убедитесь в том, что папка npm обладает подходящими разрешениями (обязательно разберитесь в том, что делает эта команда, прежде чем выполнять ее):

```
$ sudo chown -R $USER /usr/local/{share/man,bin,lib/node,include/node}
```

Установка Express

Express является такой же локальной зависимостью, как и React; это означает, что она должна быть установлена в каждом проекте. Установить Express можно только с помощью `npm`:

```
npm i express@4 -S
```

Ключ `-S` добавляет запись в package.json.

Этот раздел не дает вам исчерпывающей информации о Express.js, но он познакомит вас с самым распространенным веб-фреймворком Node.js. Сначала установите его командой `npm`:

```
$ npm install express@4.13.3
```

Обычно вы создаете серверный файл index.js, app.js или server.js, который позднее будет запущен командой `node` (например, `node index.js`). Файл состоит из следующих частей:

- импортирование;
- конфигурации;
- промежуточные модули;
- маршруты;
- обработчики ошибок;
- запуск.

Раздел импортирования тривиален. В нем включаются зависимости и создаются экземпляры объектов. Например, чтобы импортировать фреймворк Express.js и создать экземпляр, напишите следующие строки:

```
var express = require('express')
var app = express()
```

В разделе конфигураций конфигурации задаются вызовом `app.set()`, у которого в первом аргументе передается строка, а во втором — значение. Например, чтобы установить шаблонизатор Jade, используйте конфигурацию `view engine`:

```
app.set('view engine', 'jade')
```

В следующем разделе настраиваются промежуточные модули — аналоги плагинов. Например, чтобы включить предоставление статических ассетов, используйте промежуточный модуль `static`:

```
app.use(express.static(path.join(__dirname, 'public')))
```

В самой важной части определяются маршруты с использованием паттерна `app.NAME()`. Например, синтаксис конечной точки `GET /rooms` из ch20/autocomplete выглядит так:

```
app.get('/rooms', function(req, res, next) {
  req.rooms.find({}, {sort: {_id: -1}}).toArray(function(err, docs){
    if (err) return next(err)
    return res.json(docs)
  })
})
```

Обработчики ошибок похожи на промежуточные модули:

```
var errorHandler = require('errorhandler')
app.use(errorHandler)
```

Наконец, чтобы запустить приложение, выполните `listen()`:

```
http.createServer(app).listen(portNumber, callback)
```

Конечно, возможности Express.js отнюдь не ограничиваются этим кратким введением, иначе я бы не написал 350-страничную книгу об этом фреймворке («Pro Express.js»; Apress, 2014, http://proexpressjs.com)! Если вы захотите получить информацию от другого автора, могу порекомендовать книгу «Express in Action» Эвана М. Хана (Evan M. Hahn) (Manning, 2016, www.manning.com/books/express-in-action). Это мощный, но гибкий фреймворк, который настраивается без особого волшебства.

Если вы не являетесь специалистом по построению приложений Express.js или если вы уже знаете, как это делается, но хотите освежить информацию в памяти, просмотрите сводку Express.js из приложения B или ее графическую версию на сайте книги http://reactquickly.co/resources.

Установка Bootstrap

Twitter Bootstrap можно загрузить с официального веб-сайта: http://getbootstrap.com. В этой книге используется версия 3.3.5. Вы можете выбрать из нескольких вариантов:

- Загрузить архив минифицированного кода JavaScript и стилевых файлов без документации, готовый к использованию без изменений: https://github.com/twbs/bootstrap/releases/download/v3.3.5/bootstrap-3.3.5-dist.zip.

- Загрузить исходный код в Less (https://github.com/twbs/bootstrap/archive/v3.3.5.zip) или Sass (https://github.com/twbs/bootstrap-sass/archive/v3.3.5.tar.gz). Этот вариант идеально подходит для настройки.
- Загрузить по ссылке из CDN. Этот вариант более эффективен благодаря кэшированию, но для него необходим интернет.
- Установить Bootstrap с использованием Bower.
- Установить Bootstrap с использованием npm.
- Установить Bootstrap с использованием Composer.
- Создать собственную версию Bootstrap, выбрав только те компоненты, которые вам необходимы: http://getbootstrap.com/customize.
- Использовать тему Bootstrap для получения заменяемого оформления без значительной работы. Например, Bootswatch предоставляет темы Bootstrap по адресу https://bootswatch.com.

Чтобы загрузить Bootstrap по ссылке из CDN, включите в файл HTML следующие теги:

```
<!-- Новейшая откомпилированная и минифицированная разметка CSS -->
<link rel="stylesheet"
  href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.5/css/bootstrap.min.css">

<!-- Необязательная тема -->
<link rel="stylesheet"
href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.5/css/
➥ bootstrap-theme.min.css">

<!-- Новейший откомпилированный и минифицированный JavaScript -->
<script src=
  "https://maxcdn.bootstrapcdn.com/bootstrap/3.3.5/js/bootstrap.min.js">
  ➥ </script>
```

Для установки Bower, npm и Composer выполните следующие команды терминала, соответственно, в папке проекта (для каждого менеджера пакетов):

```
$ bower install bootstrap
$ npm install bootstrap
$ composer require twbs/bootstrap
```

За дополнительной информацией обращайтесь по адресу http://getbootstrap.com/getting-started.

Установка Browserify

Browserify позволяет упаковывать модули npm в пакеты клиентской части, готовые к использованию в браузере. По сути, любой модуль npm (обычно только для Node) можно превратить в модуль клиентской части.

ПРИМЕЧАНИЕ Если вы используете Webpack, вам не понадобится Browserify.

Сначала установите Browserify из `npm`:

```
$ npm install -g browserify
```

Для примера используем ch16/jest. Перейдите в эту папку и создайте файл script.js для включения библиотеки generate-password.js. Содержимое script.js может быть минимальным:

```
var generatePassword = require('generate-password')
console.log(generatePassword())
console.log(generatePassword())
```

Сохраните script.js и выполните следующую команду в терминале или командной строке:

```
$ browserify script.js -o bundle.js
```

Просмотрите файл bundle.js или включите его в index.html:

```
<script src="bundle.js"></script>
```

Откройте файл index.html в браузере и просмотрите консоль — на ней будут выведены два случайных пароля. Исходный код размещается в папке ch16/jest.

Установка MongoDB

Самый простой способ установить MongoDB — открыть адрес www.mongodb.org/downloads#production и выбрать пакет, подходящий для вашей системы.

В macOS используйте `brew` и выполните следующие команды:

```
$ brew update
$ brew install mongodb
```

Не выполняйте глобальную установку `mongodb` командой npm. Это драйвер, а не база данных, поэтому он должен располагаться с другими зависимостями в локальной папке node_modules.

В этой книге используется версия 3.0.6; используйте более новые (или старые) версии на свой собственный риск. Эти версии не были протестированы для примеров книги.

Чаще всего необходимо создать папку /data/db с соответствующими разрешениями. Вы можете сделать это или передать любую другую папку командой `mongod` с параметром `--dbpath`, например:

```
$ mongod --dbpath ./data
```

После того как база данных заработает (`mongod`), поэкспериментируйте с кодом в оболочке (`mongo`):

```
$ mongo
> 1+1
> use autocomplete
> db.rooms.find()
```

Сводка часто используемых команд оболочки:

- `> show dbs` — выводит список баз данных на сервере.
- `> use DB_NAME` — выбирает базу данных `DB_NAME`.
- `> show collections` — выводит список коллекций в выбранной базе данных.
- `> db.COLLECTION_NAME.find()` — выполняет запрос на поиск элементов в коллекции с именем `COLLECTION_NAME`.
- `> db.COLLECTION_NAME.find({"_id": ObjectId("549d9a3081d0f07866fdaac6")})` — выполняет запрос на поиск в коллекции с именем `COLLECTION_NAME` элемента с идентификатором `549d9a3081d0f07866fdaac6`.
- `> db.COLLECTION_NAME.find({"email": /gmail/})` — выполняет запрос на поиск в коллекции с именем `COLLECTION_NAME` элемента со свойством `email`, соответствующего `/gmail`.
- `> db.COLLECTION_NAME.update(QUERY_OBJECT, SET_OBJECT)` — выполняет запрос на обновление коллекции с именем `COLLECTION_NAME`: элементы, соответствующие критерию `QUERY_OBJECT`, заменяются на `SET_OBJECT`.
- `> db.COLLECTION_NAME.remove(QUERY_OBJECT)` — выполняет запрос на удаление из коллекции с именем `COLLECTION_NAME` элементов, соответствующих критерию `QUERY_OBJECT`.
- `> db.COLLECTION_NAME.insert(OBJECT)` — добавляет `OBJECT` в коллекцию с именем `COLLECTION_NAME`.

Просмотрите краткую сводку MongoDB из приложения Г или ее графическую версию на сайте книги http://reactquickly.co/resources. Кроме наиболее часто используемых команд MongoDB в нее включены методы Mongoose (Node.js ODM). Пользуйтесь!

Использование Babel для компиляции JSX и ES6

Babel в основном предназначается для ES6+/ES2015+, но также может преобразовывать JSX в JavaScript. Используя Babel с React, вы сможете пользоваться расширенными возможностями ES6 для повышения эффективности разработки.

Спецификация ES6 завершена, но ее возможности, а также возможности будущих версий ECMAScript могут лишь частично поддерживаться некоторыми браузерами. Чтобы использовать такие новые возможности, как ES Next (https://github.com/esnext/esnext), или использовать ES6 в старых браузерах (IE9), вам понадобится компилятор, такой как Babel (https://babeljs.io). Вы можете запускать его автономно или использовать с системой сборки.

Чтобы использовать Babel как автономную программу командной строки, сначала создайте новую папку. Предполагая, что в системе установлены Node.js и npm, выполните следующую команду для создания package.json:

```
$ npm init
```

Откройте файл package.json и добавьте строки `babel` в JSON. Вы можете разместить их в любом порядке при условии, что `babel` является свойством верхнего уровня. Тем самым вы говорите Babel использовать React и JSX для преобразования исходных файлов. Эта настройка называется *конфигурацией* (preset). Без нее Babel CLI работать не будет:

```
"babel": {
"presets": ["react"]
},
```

Установите Babel CLI v6.9.0 и конфигурацию React v6.5.0 из npm. В терминале, оболочке или командной строке выполните следующие команды:

```
$ npm i babel-cli@6.9.0 --save-dev
$ npm i babel-preset-react@6.5.0 --save-dev
```

Для проверки версии можно использовать следующую команду:

```
$ babel --version
```

Существуют плагины Babel для Grunt, Gulp и Webpack (http://babeljs.io/docs/setup). Рассмотрим пример для Gulp. Установите плагин:

```
$ npm install --save-dev gulp-babel
```

В файле gulpfile.js определите задачу сборки, которая компилирует src/app.js в папку build:

```
var gulp = require('gulp'),
  babel = require('gulp-babel')

gulp.task('build', function () {
  return gulp.src('src/app.js')
    .pipe(babel())
    .pipe(gulp.dest('build'))
})
```

За дополнительной информацией о Webpack и Babel обращайтесь к главе 12.

Node.js и ES6

Вы можете компилировать файлы Node.js инструментом сборки или воспользоваться автономным модулем Babel `babel-core`. Установите его следующей командой:

```
$ npm install --save-dev babel-core@6
```

Затем в Node.js вызовите следующую функцию:

```
require('babel-core').transform(es5Code, options)
```

Автономный браузер Babel

В Babel v5.x был включен файл автономного браузера, который может использоваться для внутрибраузерных преобразований (только на стадии разработки). В 6.x он был исключен, но разработчики создали модуль `babel-standalone`, чтобы исправить это упущение (https://github.com/Daniel15/babel-standalone).

Вы можете использовать эту версию или файлы из более старой версии, например из Cloudflare CDN:

- неминифицированная версия — http://mng.bz/K1b9;
- минифицированная версия — http://mng.bz/sM59.

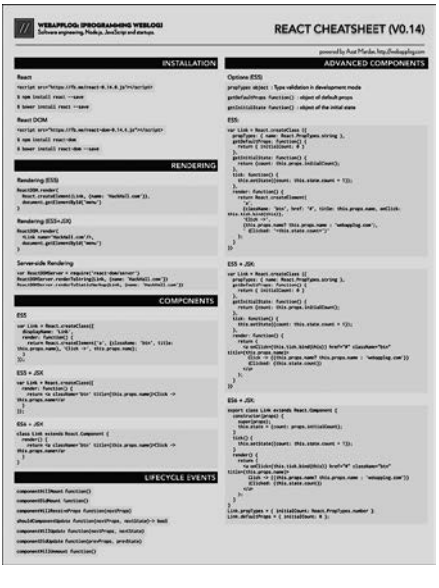
Также возможно построить собственный файл автономного браузера таким инструментом, как Gulp или Webpack. В этом случае вы сможете отобрать только те составляющие, которые вам нужны, например плагин преобразования React и конфигурации ES2015.

Б Шпаргалки по React

Когда вы занимаетесь разработкой собственных проектов, проводить поиск документации React и API в интернете или возвращаться к предыдущим главам книги для того, чтобы найти информацию об одном методе, было бы неэффективно. Если вы хотите сэкономить время и избежать лишних блужданий в интернете, эта щшпаргалка поможет вам быстро найти нужную информацию.

ДОКУМЕНТ PDF, ГОТОВЫЙ К ПЕЧАТИ

Кроме текстовой версии, приведенной в приложении, я создал красивую, готовую к печати версию документа в формате PDF. Вы можете загрузить ее по адресу http://reactquickly.co/resources.



Установка

React

- `<script src="https://unpkg.com/react@15/dist/react.js"></script>`
- `$ npm install react --save`
- `$ bower install react --save`

React DOM

- `<script src="https://unpkg.com/react-dom@15/dist/react-dom.js"></script>`
- `$ npm install react-dom`
- `$ bower install react-dom --save`

Рендеринг

ES5

```
ReactDOM.render(
    React.createElement(
      Link,
      {href: 'https://Node.University'}
    )
  ),
  document.getElementById('menu')
)
```

ES5+JSX

```
ReactDOM.render(
  <Link href='https://Node.University'/>,
  document.getElementById('menu')
)
```

Рендеринг на стороне сервера

```
const ReactDOMServer = require('react-dom/server')
ReactDOMServer.renderToString(Link, {href: 'https://Node.University'})
ReactDOMServer.renderToStaticMarkup(Link, {href: 'https://Node.University'})
```

Компоненты

ES5

```
var Link = React.createClass({
  displayName: 'Link',
  render: function() {
    return React.createElement('a',
      {className: 'btn', href: this.props.href}, 'Click ->', this.props.href)
  }
})
```

ES5 + JSX

```
var Link = React.createClass({
  render: function() {
    return <a className='btn' href={this.props.href}>Click ->
      this.props.href</a>
  }
})
```

ES6 + JSX

```
class Link extends React.Component {
  render() {
    return <a className='btn' href={this.props.href}>Click ->
      this.props.href</a>
  }
}
```

Расширенные возможности компонентов

Варианты (ES5)

Проверка типов в режиме разработки — объект `propTypes`.

Объект свойств по умолчанию — функция `getDefaultProps()`.

Объект исходного состояния — функция `getInitialState()`.

ES5

```
var Link = React.createClass ({
  propTypes: { href: React.PropTypes.string },
  getDefaultProps: function() {
    return { initialCount: 0 }
  },
  getInitialState: function() {
    return {count: this.props.initialCount}
  },
  tick: function() {
```

```
    this.setState({count: this.state.count + 1})
  },
  render: function() {
    return React.createElement(
      'a',
      {className: 'btn', href: '#', href: this.props.href,
        onClick: this.tick.bind(this)},
      'Click ->',
      (this.props.href ? this.props.href : 'https://webapplog.com'),
      ' (Clicked: ' + this.state.count+')'
    )
  }
})
```

ES5 + JSX

```
var Link = React.createClass ({
  propTypes: { href: React.PropTypes.string },
  getDefaultProps: function() {
    return { initialCount: 0 }
  },
  getInitialState: function() {
    return {count: this.props.initialCount};
  },
  tick: function() {
    this.setState({count: this.state.count + 1})
  },
  render: function() {
    return (
      <a onClick={this.tick.bind(this)} href="#" className="btn"
        href={this.props.href}>
        Click -> {(this.props.href ? this.props.href :
        ➥ 'https://webapplog.com')}
        (Clicked: {this.state.count})
      </a>
    )
  }
})
```

ES6 + JSX

```
export class Link extends React.Component {
  constructor(props) {
    super(props);
    this.state = {count: props.initialCount};
  }
  tick() {
    this.setState({count: this.state.count + 1});
  }
  render() {
    return (
      <a onClick={this.tick.bind(this)} href="#" className="btn"
        href={this.props.href}>
```

```
        Click -> {(this.props.href ? this.props.href :
          'https://webapplog.com')}
        (Clicked: {this.state.count})
      </a>
    )
  }
}
Link.propTypes = { initialCount: React.PropTypes.number }
Link.defaultProps = { initialCount: 0 }
```

События жизненного цикла

- componentWillMount function()
- componentDidMount function()
- componentWillReceiveProps function(nextProps)
- shouldComponentUpdate function(nextProps, nextState) bool
- componentWillUpdate function(nextProps, nextState)
- componentDidUpdate function(prevProps, prevState)
- componentWillUnmount function()

Последовательность событий жизненного цикла (по http://react.tips)

Подключение	Обновление свойств компонента	Обновление состояния компонента	Использование forceUpdate()	Отключение
getDefaultProps() getInitialState() componentWillMount()				
	componentWillReceiveProps()			
	shouldComponentUpdate()	shouldComponentUpdate()		
	componentWillUpdate()	componentWillUpdate()	componentWillUpdate()	
render()	render()	render()	render()	
	componentDidUpdate()	componentDidUpdate()	componentDidUpdate()	
componentDidMount()				
				componentWillUnmount()

Специальные свойства

- `key` — уникальный идентификатор элемента для преобразования массивов/списков в хеши для повышения быстродействия. Пример: `key={id}`.
- `ref` — ссылка на элемент в синтаксисе `this.refs.NAME`. Например, `ref="email"` создаст узел DOM `this.refs.email` или `ReactDOM.findDOMNode(this.refs.email)`.
- `style` — получает объект стилей CSS в «верблюжьем регистре» вместо строки; является неизменяемым с версии 0.14. Пример: `style={{color: red}}`.
- `className` — атрибут класса HTML. Пример: `className="btn"`.
- `htmlFor` — HTML для атрибута. Пример: `htmlFor="email"`.
- `dangerouslySetInnerHTML` — заполняет внутреннюю разметку низкоуровневой разметкой HTML, предоставляя объект с ключом `__html`.
- `children` — задает контент элемента через `this.props.children`. Пример: `this.props.children[0]`.
- `data-NAME` — нестандартный атрибут. Пример: `data-tooltip-text="..."`.

propTypes

Типы, доступные в `React.PropTypes`:

- `any`
- `array`
- `bool`
- `element`
- `func`
- `node`
- `number`
- `object`
- `string`

Чтобы сделать свойство обязательным (только предупреждение!), присоедините `.isRequired`.

Другие методы:

- `instanceOf(constructor)`
- `oneOf(['News', 'Photos'])`
- `oneOfType([propType, propType])`

Нестандартная проверка

```
propTypes: {
  customProp: function(props, propName, componentName) {
    if (!/regExPattern/.test(props[propName])) {
      return new Error('Validation failed!');
    }
  }
}
```

Свойства и методы компонентов

Свойства

- `this.refs` — перечисляет компоненты со свойством `ref`.
- `this.props` — перечисляет свойства, переданные элементу (неизменяемые).
- `this.state` — перечисляет состояния, заданные методами `setState` и `getInitialState` (изменяемые). Не следует назначать состояние вручную с помощью `this.state=....`
- `this.isMounted` — показывает, имеет ли элемент соответствующий узел DOM.

Методы

- `setState(changes)` — заменяет состояние (частично) на `this.state` и инициирует перерисовку.
- `replaceState(newState)` — заменяет `this.state` и инициирует перерисовку.
- `forceUpdate()` — инициирует немедленную перерисовку DOM.

Дополнения React

В форме модулей npm:

- `react-addons-css-transition-group` (http://facebook.github.io/react/docs/animation.html)
- `react-addons-perf` (http://facebook.github.io/react/docs/perf.html)
- `react-addons-test-utils` (http://facebook.github.io/react/docs/test-utils.html)
- `react-addons-pure-render-mixin` (http://facebook.github.io/react/docs/pure-render-mixin.html)
- `react-addons-linked-state-mixin` (http://facebook.github.io/react/docs/two-way-binding-helpers.html)
- `react-addons-clone-with-props`

- `react-addons-create-fragment`
- `react-addons-css-transition-group`
- `react-addons-linked-state-mixin`
- `react-addons-pure-render-mixin`
- `react-addons-shallow-compare`
- `react-addons-transition-group`
- `react-addons-update` (http://facebook.github.io/react/docs/update.html)

Компоненты React

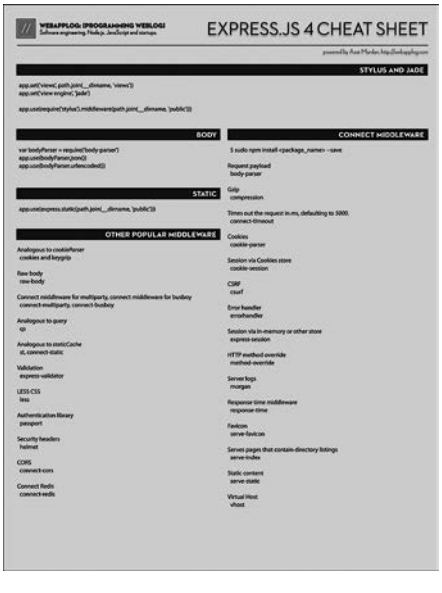
- Списки компонентов React — https://github.com/brillout/awesome-react-components и http://devarchy.com/react-components
- Material-UI — компоненты React для Material Design (http://material-ui.com)
- React Toolbox — компоненты React, реализующие спецификацию Google Material Design (http://react-toolbox.com)
- JS.Coach — подборка пакетов JS с открытым кодом (в основном React) (https://js.coach)
- React Rocks — подборка компонентов React (https://react.rocks)
- Khan Academy — подборка компонентов React, пригодных для повторного использования (https://khan.github.io/react-components)
- ReactJSX.com — реестр компонентов React (http://reactjsx.com)

B Краткая сводка Express.js

Когда вы занимаетесь разработкой собственных проектов, проводить поиск документации React и API в интернете или возвращаться к предыдущим главам книги для того, чтобы найти информацию об одном методе, было бы неэффективно. Если вы хотите сэкономить время и избежать лишних блужданий в интернете, эта краткая сводка Express поможет вам быстро найти нужную информацию.

ДОКУМЕНТ PDF, ГОТОВЫЙ К ПЕЧАТИ

Кроме текстовой версии, приведенной в приложении, я создал красивую, готовую к печати версию документа в формате PDF. Вы можете загрузить ее по адресу http://reactquickly.co/resources.



Установка Express.js

- `$ sudo npm install express` — выполняет локальную установку новейшей версии Express.js.
- `$ sudo npm install express@4.2.0 --save` — выполняет локальную установку Express.js v4.2.0 и сохраняет в файле package.json.
- `$ sudo npm instal -g express-generator@4.0.0` — выполняет установку генератора командной строки Express.js v4.0.0.

Генератор

Использование

```
$ express [options] [dir]
```

Параметры

- `-h` — выводит информацию об использовании.
- `-V` — выводит номер версии `express-generator`.
- `-e` — добавляет поддержку ядра EJS; если ключ не указан, по умолчанию используется Jade.
- `-H` — добавляет поддержку ядра hogan.js.
- `-c <library>` — добавляет поддержку CSS для <library> (`less|stylus|compass`); если параметр `-c <library>` не указан, по умолчанию используется простая разметка CSS.
- `-f` — генерирует в непустой каталог.

Основы

- `var express = require('express')` — включает модуль.
- `var app = express()` — создает экземпляр.
- `app.listen(portNumber, callback)` — запускает сервер Express.js.
- `http.createServer(app).listen(portNumber, callback)` — запускает сервер Express.js.
- `app.set(key, value)` — задает значение свойства по ключу.
- `app.get(key)` — получает значение свойства по ключу.

Маршруты и команды HTTP

- `app.get(urlPattern, requestHandler[, requestHandler2, ...])` — обрабатывает GET-запросы.
- `app.post (urlPattern, requestHandler[, requestHandler2, ...])` — обрабатывает POST-запросы.
- `app.put (urlPattern, requestHandler[, requestHandler2, ...])` — обрабатывает PUT-запросы.
- `app.delete (urlPattern, requestHandler[, requestHandler2, ...])` — обрабатывает DELETE-запросы.
- `app.all (urlPattern, requestHandler[, requestHandler2, ...])` — обрабатывает все запросы.
- `app.param ([name,] callback)` — обрабатывает параметры URL.
- `app.use ([urlPattern,] requestHandler[, requestHandler2, ...])` — применяет промежуточные модули.

Запросы

- `request.params` — параметр функции промежуточной обработки.
- `request.param` — извлекает один параметр.
- `request.query` — извлекает строку запроса.
- `request.route` — возвращает строку маршрута.
- `request.cookies` — обращается к данным cookie; требует cookie-parser.
- `request.signedCookies` — обращается к подписанным cookie; требует requires cookie-parser.
- `request.body` — читает основные данные; требует body-parser.

Сокращенные обозначения заголовков

- `request.get(headerKey)` — читает `Value` для ключа заголовка.
- `request.accepts(type)` — проверяет, приемлем ли тип.
- `request.acceptsLanguage(language)` — проверяет язык.
- `request.acceptsCharset(charset)` — проверяет кодировку.
- `request.is(type)` — проверяет тип.

- `request.ip` — читает IP-адрес.
- `request.ips` — читает IP-адреса (с включенным trust-proxy).
- `request.path` — читает путь URL.
- `request.host` — обращается к хосту без номера порта.
- `request.fresh` — проверяет свежесть данных.
- `request.stale` — проверяет устаревание.
- `request.xhr` — проверяет запросы в стиле XHR/AJAX.
- `request.protocol` — возвращает протокол HTTP.
- `request.secure` — проверяет, используется ли протокол HTTPS.
- `request.subdomains` — читает массив субдоменов.
- `request.originalUrl` — читает исходный URL-адрес.

Ответ

- `response.redirect(status, url)` — перенаправляет запрос.
- `response.send(status, data)` — отправляет ответ.
- `response.json(status, data)` — отправляет данные JSON и создает соответствующие заголовки.
- `response.sendfile(path, options, callback)` — отправляет файл.
- `response.render(templateName, locals, callback)` — рендерит шаблон.
- `response.locals` — передает данные шаблону.

Сигнатуры обработчиков

- `function(request, response, next) {}` — сигнатура обработчика запроса.
- `function(error, request, response, next) {}` — сигнатура обработчика ошибок.

Stylus и Jade

Установка Jade и Stylus:

```
$ npm i -SE stylus jade
```

Применение шаблонизатора Jade:

```
app.set('views', path.join(__dirname, 'views'))
app.set('view engine', 'jade')
```

Применение процессора CSS Stylus:

```
app.use(require('stylus').middleware(path.join(__dirname, 'public')))
```

Тело

```
var bodyParser = require('body-parser')
app.use(bodyParser.json())
app.use(bodyParser.urlencoded({
    extended: true
}))
```

Статические файлы

```
app.use(express.static(path.join(__dirname, 'public')))
```

Подсоединение промежуточных модулей

```
$ sudo npm install <имя_пакета> --save
```

- `body-parser` (https://github.com/expressjs/body-parser) — обращается к данным запроса.
- `compression` (https://github.com/expressjs/compression) — выполняет сжатие с использованием Gzip.
- `connect-timeout` (https://github.com/expressjs/timeout) — закрывает запросы после истечения заданного периода времени.
- `cookie-parser` (https://github.com/expressjs/cookie-parser) — разбирает и читает cookie.
- `cookie-session` (https://github.com/expressjs/cookie-session) — использует сеанс через хранилище cookie.
- `csurf` (https://github.com/expressjs/csurf) — генерирует маркер (token) для межсайтовой подделки запросов CSRF (Cross-Site Request Forgery).
- `errorhandler` (https://github.com/expressjs/errorhandler) — использует обработчики ошибок стадии разработки.
- `express-session` (https://github.com/expressjs/session) — использует сеанс через хранилище в памяти или другое хранилище.
- `method-override` (https://github.com/expressjs/method-override) — переопределяет методы HTTP.
- `morgan` (https://github.com/expressjs/morgan) — выводит лог сервера.
- `response-time` (https://github.com/expressjs/response-time) — выводит время ответа.
- `serve-favicon` (https://github.com/expressjs/serve-favicon) — предоставляет значок сайта.

- `serve-index` (https://github.com/expressjs/serve-index) — предоставляет список содержимого каталога и файлов в форме файлового сервера.
- `serve-static` (https://github.com/expressjs/serve-static) — предоставляет статический контент.
- `vhost` (https://github.com/expressjs/vhost) — использует виртуальный хост.

Другие популярные промежуточные модули

- `cookies` (https://github.com/jed/cookies) и `keygrip` (https://github.com/jed/keygrip) — разбирает cookie (по аналогии с `cookie-parser`).
- `raw-body` (https://github.com/stream-utils/raw-body) — использует низкоуровневые данные/тело.
- `connect-multiparty` (https://github.com/superjoe30/connect-multiparty) — обрабатывает операции отправки файлов.
- `qs` (https://github.com/ljharb/qs) — разбирает строки запросов с объектами и массивами как значениями.
- `st` (https://github.com/isaacs/st) и `connect-static` (https://github.com/andrewrk/connect-static) — предоставляет статические файлы (по аналогии с `staticCache`).
- `express-validator` (https://github.com/ctavan/express-validator) — выполняет проверку данных.
- `less` (https://github.com/emberfeather/less.js-middleware) — обрабатывает файлы LESS в CSS.
- `passport` (https://github.com/jaredhanson/passport) — аутентифицирует запрос.
- `helmet` (https://github.com/evilpacket/helmet) — задает заголовки безопасности.
- `connect-cors` (https://npmjs.com/package/cors) — включает совместное использование ресурсов между разными источниками (CORS, Cross-Origin Resource Sharing).
- `connect-redis` (http://github.com/visionmedia/connect-redis) — подключается к Redis.

Ресурсы

- Бесплатный интернет-курс Express Foundation, https://node.university/p/express-foundation
- Pro Express (Apress, 2014) — моя подробная книга о Express.js, http://proexpressjs.com
- Сообщения об Express.js в моем блоге, https://webapplog.com/tag/express-js

Шпаргалка по MongoDB и Mongoose

Когда вы занимаетесь разработкой собственных проектов, проводить поиск документации React и API в интернете или возвращаться к предыдущим главам книги для того, чтобы найти информацию об одном методе, было бы неэффективно. Если вы хотите сэкономить время и избежать лишних блужданий в интернете, эта шпаргалка по MongoDB поможет вам быстро найти нужную информацию.

ДОКУМЕНТ PDF, ГОТОВЫЙ К ПЕЧАТИ

Кроме текстовой версии, приведенной в приложении, я создал красивую, готовую к печати версию документа в формате PDF. Вы можете загрузить ее по адресу http://reactquickly.co/resources.

MongoDB

- `$ mongod` — запускает сервер MongoDB (*localhost:27017*).
- `$ mongo` (по умолчанию подключается к локальному серверу) — открывает консоль MongoDB.

Консоль MongoDB

- `> show dbs` — выводит список баз данных на сервере.
- `> use DB_NAME` — выбирает базу данных с именем `DB_NAME`.
- `> show collections` — выводит список коллекций в выбранной базе данных.
- `> db.COLLECTION_NAME.find()` — выводит запрос на поиск элементов в коллекции с именем `COLLECTION_NAME`.
- `> db.COLLECTION_NAME.find({"_id" –ObjectId("549d9a3081d0f07866fdaac6")})` — выполняет запрос на поиск в коллекции с именем `COLLECTION_NAME` элемента с идентификатором 549d9a3081d0f07866fdaac6.
- `> db.COLLECTION_NAME.find({"email": /gmail/})` — выполняет запрос на поиск в коллекции с именем `COLLECTION_NAME` элемента со свойством `email`, соответствующего `/gmail`.
- `> db.COLLECTION_NAME.update(QUERY_OBJECT, SET_OBJECT)` — выполняет запрос на обновление коллекции с именем `COLLECTION_NAME`: элементы, соответствующие критерию `QUERY_OBJECT`, заменяются на `SET_OBJECT`.
- `> db.COLLECTION_NAME.remove(QUERY_OBJECT)` — выполняет запрос на удаление из коллекции с именем `COLLECTION_NAME` элементов, соответствующих критерию `QUERY_OBJECT`.
- `> db.COLLECTION_NAME.insert(OBJECT)` — добавляет `OBJECT` в коллекцию с именем `COLLECTION_NAME`.

Установка Mongoose

- `$ sudo npm install mongoose` — выполняет локальную установку новейшей версии Mongoose.
- `$ sudo npm install mongoose@3.8.20 --save` — выполняет локальную установку Mongoose v3.8.20 и сохраняет ее в файле package.json.

Простейший вариант использования Mongoose

```
var mongoose = require('mongoose')
var dbUri = 'mongodb://localhost:27017/api'
var dbConnection = mongoose.createConnection(dbUri)
var Schema = mongoose.Schema
```

```
var postSchema = new Schema ({
  title: String,
  text: String
})
var Post = dbConnection.model('Post', postSchema, 'posts')
Post.find({},function(error, posts){
  console.log(posts)
  process.exit(1)
})
```

Схема Mongoose

- String
- Boolean
- Number
- Date
- Array
- Buffer
- Schema.Types.Mixed
- Schema.Types.ObjectId

Пример Mongoose с использованием CRUD (Create/Read/Update/Delete)

```
// Создание
var post = new Post({title: 'a', text: 'b')
post.save(function(error, document){
  ...
})

// Чтение
Post.findOne(criteria, function(error, post) {
  ...
})

// Обновление
Post.findOne(criteria, function(error, post) {
  post.set()
  post.save(function(error, document){
    ...
  })
})

// Удаление
Post.findOne(criteria, function(error, post) {
  post.remove(function(error){
    ...
  })
})
```

Методы модели Mongoose

- `find(criteria, [fields], [options], [callback])`, где `callback` получает аргументы `error` и `documents`, — ищет документ.
- `count(criteria, [callback]))`, где `callback` получает аргументы `error` и `count`, — возвращает количество документов, соответствующих заданному критерию.
- `findById(id, [fields], [options], [callback])`, где `callback` получает аргументы `error` и `documents`, — возвращает один документ по идентификатору.
- `findByIdAndUpdate(id, [update], [options], [callback])` — выполняет метод MongoDB `findAndModify()` для обновления документа по идентификатору.
- `findByIdAndRemove(id, [options], [callback])` — выполняет метод MongoDB `findAndModify()` для удаления документа по идентификатору.
- `findOne(criteria, [fields], [options], [callback])`, где `callback` получает аргументы `error` и `documents`, — возвращает один документ.
- `findOneAndUpdate ([criteria], [update], [options], [callback])` — выполняет метод MongoDB `findAndModify()` для обновления документ(-ов).
- `findOneAndRemove(id, [update], [options], [callback])` — выполняет метод MongoDB `findAndModify()` для удаления документа.
- `update(criteria, update, [options], [callback])`, где `callback` получает аргументы `error` и `count`, — обновляет документы.
- `create(doc(s), [callback])`, где `callback` получает аргументы `error` и `doc(s)` — создает объект документа и сохраняет его в базе данных.
- `remove(criteria, [callback])`, где `callback` получает аргумент `error`, — удаляет документы.

Методы документов Mongoose

- `save([callback])`, где `callback` получает аргументы `error`, `doc` и `count`, — сохраняет документ.
- `set(path, val, [type], [options])` — задает значение свойства документа.
- `get(path, [type])` — получает значение свойства.
- `isModified([path])` — проверяет, было ли изменено значение свойства.
- `populate([path], [callback])` — заполняет ссылку.
- `toJSON(options)` — получает JSON из документа.
- `validate(callback)` — проверяет документ.

ES6 для успеха

В этом приложении приведено краткое введение в ES6. В нем описаны 10 лучших возможностей нового поколения самого популярного языка программирования — JavaScript:

1. Параметры по умолчанию.
2. Шаблонные литералы.
3. Многострочные строки.
4. Деструктурирующее присваивание.
5. Расширенные объектные литералы.
6. «Стрелочные» функции.
7. Обещания.
8. Конструкции с блочной областью видимости: let и const.
9. Классы.
10. Модули.

> **ПРИМЕЧАНИЕ** Этот список в высшей степени субъективен. Он никоим образом не умаляет полезности других возможностей ES6, которые не попали в список только потому, что я хотел ограничить его 10 пунктами.

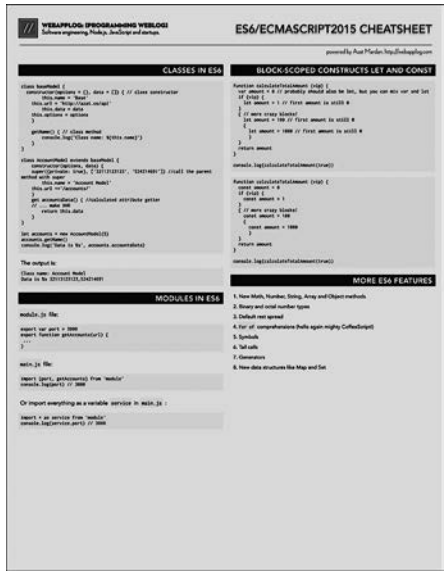
Параметры по умолчанию

Помните те времена, когда для определения параметров по умолчанию нам приходилось использовать конструкции следующего вида?

```
var link = function (height, color, url) {
    var height = height || 50
    var color = color || 'red'
    var url = url || 'http://azat.co'
    ...
}
```

ДОКУМЕНТ PDF, ГОТОВЫЙ К ПЕЧАТИ

Кроме этого обзора, я создал бесплатную, красивую, готовую к печати краткую сводку ES6/ES2015. Документ в формате PDF можно загрузить по адресу http://reactquickly.co/resources.



Этот способ хорошо работал, пока значение было отлично от 0. С 0 могла возникнуть ошибка: вместо фактического значения использовалось жестко фиксированное, потому что значение 0 в JavaScript имеет тип false. Конечно, кому придет в голову использовать значение 0 (`#sarcasmfont`)? Поэтому мы игнорировали этот недостаток и использовали логическую операцию OR. Забудьте об этом! В ES6 дефолтное значение размещается прямо в сигнатуре функции:

```
var link = function(height = 50, color = 'red', url = 'http://azat.co') {
  ...
}
```

Этот синтаксис напоминает синтаксис Ruby. В моем любимом CoffeeScript эта возможность также поддерживается — и поддерживалась в течение многих лет.

Шаблонные литералы

Шаблонные литералы, или интерполяция в других языках программирования, представляют собой удобный механизм вывода переменных в строках. В ES5 строку приходилось разбивать на части:

```
var name = 'Your name is'+first+''+ last + '.'
var url = 'http://localhost:3000/api/messages/' + id
```

В ES6 можно использовать новый синтаксис `${NAME}` в строках, заключенных в обратные апострофы:

```
var name = `Your name is ${first} ${last}.`
var url = `http://localhost:3000/api/messages/${id}`
```

Вас когда-нибудь интересовало, можно ли использовать синтаксис шаблонных литералов с Markdown? В Markdown обратные апострофы используются для встроенных блоков кода. И это создает проблемы! Правильное решение — использовать две, три обратные кавычки и более для кода Markdown, использующего обратные апострофы для строковых шаблонов.

Многострочные строки

Еще одно синтаксическое удобство — многострочные строки. В ES5 приходилось использовать одно из следующих решений:

```
var roadPoem = 'Then took the other, as just as fair,\n\t'
    + 'And having perhaps the better claim\n\t'
    + 'Because it was grassy and wanted wear,\n\t'
    + 'Though as for that the passing there\n\t'
    + 'Had worn them really about the same,\n\t'

var fourAgreements = 'You have the right to be you.\n\
    You can only be you when you do your best.'
```

В ES6 можно использовать обратные апострофы:

```
var roadPoem = `Then took the other, as just as fair,
    And having perhaps the better claim
    Because it was grassy and wanted wear,
    Though as for that the passing there
    Had worn them really about the same,`

var fourAgreements = `You have the right to be you.
    You can only be you when you do your best.`
```

Деструктурирующее присваивание

Понять концепцию деструктуризации будет сложнее, потому что здесь происходит нечто особенное. Допустим, у вас имеются простые команды присваивания, в которых свойства/атрибуты `house` и `mouse` ключей/объектов присваиваются переменным `house` и `mouse`:

```
var data = $('body').data(),  ◄——  data содержит свойства house и mouse
  house = data.house,
  mouse = data.mouse
```

Еще несколько примеров деструктурирующего присваивания (из Node.js):

```
var jsonMiddleware = require('body-parser').json

var body = req.body,  ◄——  body содержит свойства username и password
  username = body.username,
  password = body.password
```

В ES6 этот код ES5 можно заменить следующими командами:

```
var { house, mouse} = $('body').data()  ◄——  Здесь создаются переменные house и mouse

var {json} = require('body-parser')

var {username, password} = req.body
```

То же самое можно делать с массивами. Сюрприз!

```
var [col1, col2] = $('.column'),
[line1, line2, line3, , line5] = file.split('\n')
```

Первая команда присваивает элемент 0 переменной `col1`, а элемент 1 — переменной `col2`. Вторая команда (да, переменная `line4` пропущена намеренно) реализует следующее присваивание, где `fileSplitArray` — результат `file.split('\n')`:

```
var line1 = fileSplitArray[0]
var line2 = fileSplitArray[1]
var line3 = fileSplitArray[2]
var line5 = fileSplitArray[4]
```

Возможно, вы не сразу привыкнете к синтаксису деструктурирующего присваивания, но он удобен — никаких сомнений по этому поводу нет.

Расширенные объектные литералы

Новые возможности работы с объектными литералами просто невероятны! От прославленной версии JSON в ES5 мы пришли к нечто напоминающему классы в ES6.

Типичный объектный литерал ES5 с методами и атрибутами/свойствами:

```
var serviceBase = {port: 3000, url: 'azat.co'},
  getAccounts = function(){return [1,2,3]}
```

```
var accountServiceES5 = {
  port: serviceBase.port,
  url: serviceBase.url,
  getAccounts: getAccounts,
  toString: function() {
    return JSON.stringify(this.valueOf())
  },
  getUrl: function() {return "http://" + this.url + ':' + this.port},
  valueOf_1_2_3: getAccounts()
}
```

Если вы любите творческие решения, примените наследование от объекта `serviceBase`, сделав его прототипом при помощи метода `Object.create()`:

```
var accountServiceES5ObjectCreate = Object.create(serviceBase)
var accountServiceES5ObjectCreate = {
  getAccounts: getAccounts,
  toString: function() {
    return JSON.stringify(this.valueOf())
  },
  getUrl: function() {return "http://" + this.url + ':' + this.port},
  valueOf_1_2_3: getAccounts()
}
```

Я знаю, что `accountServiceES5ObjectCreate` и `accountServiceES5` *не* идентичны, потому что один объект (`accountServiceES5`) обладает свойствами объекта *proto* (рис. Д.1). Тем не менее в контексте этого примера мы будем считать их достаточно близкими. У объектных литералов ES6 существует сокращенная запись присваивания: `getAccounts: getAccounts,` превращается просто в `getAccounts` без двоеточия.

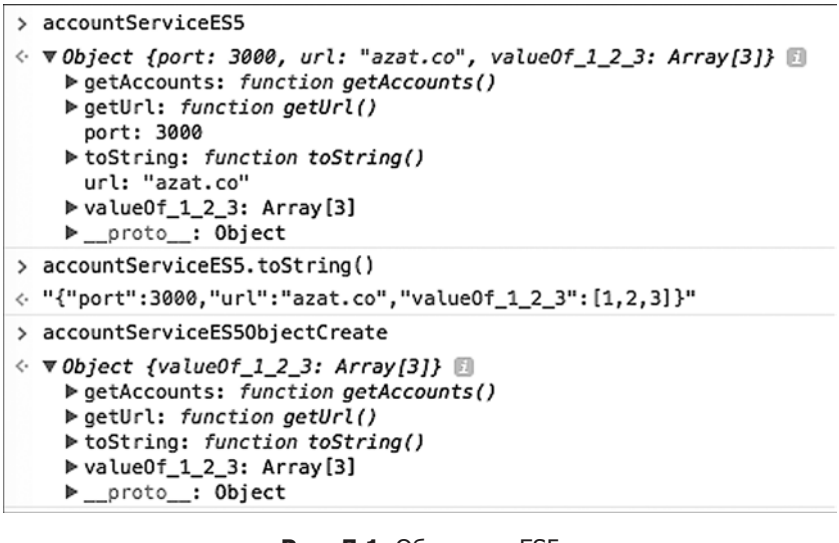


Рис. Д.1. Объекты в ES5

Кроме того, прототип задается в свойстве `__proto__`, что выглядит логично (но не в `'__proto__'` — это было бы обычное свойство):

```
var serviceBase = {port: 3000, url: 'azat.co'},
    getAccounts = function(){return [1,2,3]}
var accountService = {
    __proto__: serviceBase,
    getAccounts,
```

Кроме того, в `toString()` можно вызывать `super`:

```
toString() {
    return JSON.stringify((super.valueOf()))
  },
  getUrl() {return "http://" + this.url + ':' + this.port},
```

Вы можете динамически создавать ключи, свойства объектов и атрибуты: например, `valueOf_1_2_3` с использованием конструкции `[ 'valueOf_' + getAccounts().join('_') ]`:

```
[ 'valueOf_' + getAccounts().join('_') ]: getAccounts()
}
console.log(accountService)
```

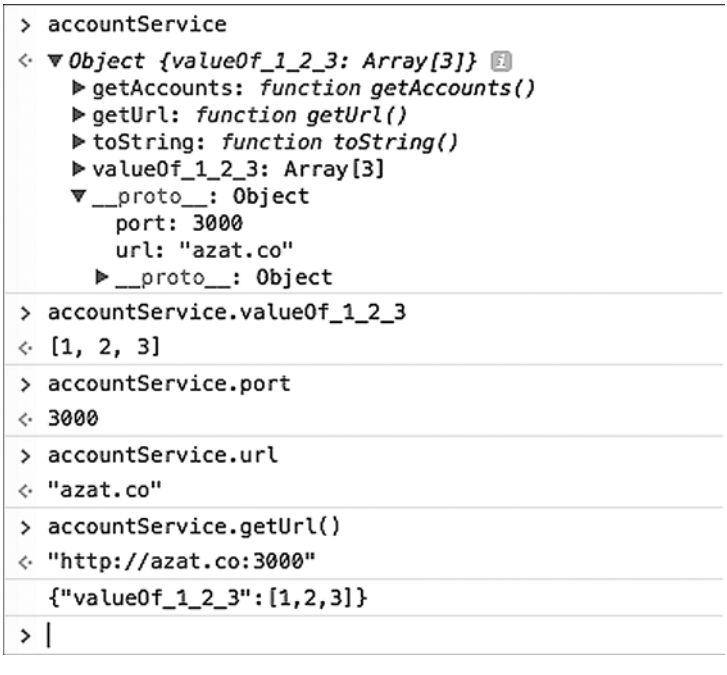


Рис. Д.2. Объектный литерал ES6 расширяет serviceBase и определяет методы и атрибуты

Объект ES6 с `__proto__` в качестве объекта `serviceBase` показан на рис. Д.2. Заметный шаг вперед по сравнению со старыми добрыми объектными литералами!

Стрелочные функции

Вероятно, из всех новых возможностей эта была для меня самой желанной. Мне нравился стрелочный синтаксис в CoffeeScript, а теперь он появился и в ES6. Прежде всего, стрелочные функции экономят место и время благодаря своей компактности:

```
const sum = (a, b, c) => {
  returna+b+c
}
```

У стрелочных функций есть и другое достоинство: они обеспечивают правильность работы `this`, что имеет такое же значение, как в контексте функции — `this` не изменяется. Изменение обычно происходит каждый раз, когда вы создаете замыкание.

Использование стрелочных функций в ES6 означает, что вам уже не нужно использовать `that = this`, `self = this`, `_this = this` и `.bind(this)`. Например, следующий код в ES5 выглядит некрасиво:

```
var _this = this
$('.btn').click(function(event){
  _this.sendData()
})
```

Код ES6 намного лучше:

```
$('.btn').click((event) => {
  this.sendData()
})
```

К сожалению, комитет ES6 решил, что поддерживать еще и «тонкие» стрелки было бы слишком хорошо, и оставил нам более громоздкий синтаксис `function()`. (Тонкие стрелки в CoffeeScript работают как обычные функции в ES5 и ES6.)

Другой пример, в котором `call` используется для передачи контекста функции `logUpperCase()` в ES5:

```
var logUpperCase = function() {
  var _this = this

  this.string = this.string.toUpperCase()
  return function () {
    return console.log(_this.string)
  }
}

logUpperCase.call({ string: 'es6 rocks' })()
```

В ES6 вам не нужно возиться с `_this`:

```
var logUpperCase = function() {
  this.string = this.string.toUpperCase()
  return () => console.log(this.string)
}

logUpperCase.call({ string: 'es6 rocks' })()
```

Старый синтаксис `function` в ES6 можно комбинировать с `=>` так, как вы считаете нужным. А когда стрелочная функция используется в однострочной команде, она становится выражением; иначе говоря, она неявно возвращает результат этой одной команды. Если строк несколько, то вам придется явно использовать `return`.

Следующий код ES5, создающий массив на основе массива `messages`:

```
var ids = ['5632953c4e345e145fdf2df8','563295464e345e145fdf2df9']
var messages = ids.map(function (value) {
  return "ID is " + value  <-- Явный возврат
});
```

в ES6 будет выглядеть так:

```
var ids = ['5632953c4e345e145fdf2df8','563295464e345e145fdf2df9']
var messages = ids.map(value => `ID is ${value}`) //  <-- Неявный возврат
```

Обратите внимание: в этом коде используются строковые шаблоны. Еще одна из моих любимых возможностей CoffeeScript!

Круглые скобки (`()`) не обязательны для одиночных параметров в сигнатуре стрелочной функции. Они необходимы только в том случае, если параметров несколько. В ES5 следующий код содержит `function()` с явной командой `return`:

```
var ids = ['5632953c4e345e145fdf2df8', '563295464e345e145fdf2df9'];
var messages = ids.map(function (value, index, list) {
  return 'ID of ' + index + ' element is ' + value + ' '  <-- Явный возврат
})
```

В более длинной версии кода в ES6 параметры заключены в круглые скобки, и используется неявный возврат:

```
var ids = ['5632953c4e345e145fdf2df8','563295464e345e145fdf2df9']
var messages = ids.map((value, index, list) =>
  `ID of ${index} element is ${value} `)  <-- Неявный возврат
```

Обещания

Обещания, или промисы (promises) традиционно были спорной темой. Было много реализаций обещаний с незначительными различиями в синтаксисе: Q, Bluebird,

Deferred.js, Vow, Avow и jQuery Deferred, и это далеко не полный список. Другие разработчики считают, что обещания не нужны: можно использовать асинхронные функции, генераторы, обратные вызовы и т. д. К счастью, в ES6 теперь имеется стандартная реализация обещаний `Promise`.

Рассмотрим тривиальный пример отложенного асинхронного выполнения с `setTimeout()`:

```
setTimeout(function(){
  console.log('Yay!')
}, 1000)
```

Этот код можно переписать в ES6 с `Promise` в следующем виде:

```
var wait1000 = new Promise(function(resolve, reject) {
  setTimeout(resolve, 1000)
}).then(function() {
  console.log('Yay!')
})
```

Также можно использовать «стрелочные» функции ES6:

```
var wait1000 = new Promise((resolve, reject)=> {
  setTimeout(resolve, 1000)
}).then(()=> {
  console.log('Yay!')
})
```

Пока что количество строк увеличилось с трех до пяти без сколько-нибудь ощутимых преимуществ. Преимущества проявляются, когда обратный вызов `setTimeout()` содержит больший объем вложенной логики. Следующий код:

```
setTimeout(function(){
  console.log('Yay!')
  setTimeout(function(){
    console.log('Wheeyee!')
  }, 1000)
}, 1000)
```

может быть переписан с обещаниями ES6 в следующем виде:

```
var wait1000 = ()=> new Promise((resolve, reject)=>
  {setTimeout(resolve, 1000)})

wait1000()
    .then(function() {
      console.log('Yay!')
      return wait1000()
  })
  .then(function() {
      console.log('Wheeyee!')
  });
```

Все еще не убеждены, что обещания лучше обычных обратных вызовов? Я тоже. Я думаю, что когда вы поймете, как работают обратные вызовы, в дополнительной сложности обещаний уже не будет необходимости. Тем не менее обещания доступны в ES6 для тех, кому они нравятся, и у них есть обратные вызовы с сообщениями о перехвате ошибок, а это удобно. За дополнительной информацией об обещаниях обращайтесь к сообщению Джеймса Нельсона (James Nelson) «Introduction to ES6 Promises: The Four Functions You Need to Avoid Callback Hell» (http://mng.bz/3OAP).

Конструкции с блочной областью видимости: let и const

Возможно, вам уже попадалась странноватая конструкция `let` в коде ES6. Она не относится к простым синтаксическим удобствам, в ней заложен более глубокий смысл. `let` — новая разновидность `var`, позволяющая ограничить область видимости переменной блоком. Блоки определяются в фигурных скобках. В ES5 блоки *никак* не влияли на переменные:

```
function calculateTotalAmount (vip) {
  var amount = 0
  if (vip) {
    var amount = 1
  }
  { // Больше блоков!
    var amount = 100
    {
      var amount = 1000
      }
  }
  return amount
}

console.log(calculateTotalAmount(true))
```

Результат равен 1000. Ого! Похоже, это серьезная ошибка. В ES6 `let` ограничивает область видимости блоком. Область видимости обычных переменных определяется границами функции:

```
function calculateTotalAmount (vip) {
  var amount = 0 // Вероятно, тоже должно быть let, но вы можете смешивать var
и let в своих программах
  if (vip) {
    let amount = 1 // Первое значение amount остается равным 0
  }
  { // Больше блоков!
    let amount = 100 // Первое значение amount остается равным 0
    {
      let amount = 1000 // Первое значение amount остается равным 0
```

```
      }
  }
  return amount
}

console.log(calculateTotalAmount(true))
```

Значение равно 0, потому что в блоке `if` тоже используется `let`. Если бы `let` не было (`amount=1`), то выражение было бы равно 1.

С `const` дело обстоит проще: создается ссылка, доступная только для чтения, и она имеет блочную область видимости, как и `let`. («Доступность только для чтения» означает, что идентификатору переменной невозможно присвоить новое значение.) `const` также работает с объектами, их свойства могут изменяться.

Допустим, у вас есть константа `url`: `const url="http://webapplog.com"`. Попытка выполнить повторное присваивание вида `const url="http://azat.co"` в большинстве браузеров завершится неудачей: хотя в документации утверждается, что `const` не подразумевает неизменяемости, при попытке изменить значение оно не изменится.

Для примера приведу допустимый набор констант, потому что они принадлежат разным блокам:

```
function calculateTotalAmount (vip) {
  const amount = 0
  if (vip) {
    const amount = 1
  }
  { // Больше блоков!
    const amount = 100
    {
      const amount = 1000
      }
  }
  return amount
}

console.log(calculateTotalAmount(true))
```

По моему скромному мнению, `let` и `const` излишне усложняют язык. Без них в языке было только одно поведение; теперь приходится рассматривать несколько сценариев.

Классы

Если вы любите объектно-ориентированное программирование, то это нововведение вам понравится. С ним писать классы и наследовать от них становится не намного сложнее, чем лайкнуть комментарий на Facebook.

В ES5 создание и использование классов было весьма непростым делом, потому что в языке не было ключевого слова `class` (оно было зарезервировано, но ничего не делало). Кроме того, многочисленные паттерны наследования — псевдоклассический[1], классический[2] и функциональный — только усиливали путаницу, подливая масла в костер войн JavaScript.

Я не буду показывать, как написать класс в ES5, потому что существует много разных модификаций. Рассмотрим пример для ES6. Класс ES6 использует прототипы, а не паттерн с функцией-фабрикой. Вот как выглядит класс `baseModel`, в котором можно определить конструктор и метод `getName()`:

```
class baseModel {
  constructor(options = {}, data = [])   {  ◄── Метод класса
    this.name = 'Base'
    this.url = 'http://azat.co/api'
    this.data = data
    this.options = options
  }
  getName() {  ◄── Конструктор класса
    console.log(`Class name: ${this.name}`)
  }
}
```

Обратите внимание: в этом коде используются значения параметров по умолчанию для `options` и `data`. Кроме того, в имена методов уже не обязательно включать слово `function` или двоеточие (`:`). Другое важное различие заключается в том, что вы не можете задавать значения свойств (`this.NAME`) по аналогии с методами. Значение свойства должно задаваться в конструкторе.

`AccountModel` наследует от `baseModel` в синтаксисе `class NAME extends PARENT_NAME`. Чтобы вызвать родительский конструктор, используйте вызов `super()` с параметрами:

```
class AccountModel extends baseModel {
  constructor(options, data) {
    super({private: true}, ['32113123123', '524214691'])
    this.name = 'Account Model'
    this.url +='/accounts/'
  }
}
```

Вызывает метод-конструктор родительского класса

Если вам захочется действовать более изобретательно, создайте get-метод следующего вида, и `accountsData` станет свойством:

```
class AccountModel extends baseModel {
  constructor(options, data) {
```

[1] См. Ilya Kantor, «Class Patterns», http://javascript.info/class-patterns.

[2] См. Douglas Crockford, «Classical Inheritance in JavaScript», www.crockford.com/javascript/inheritance.html.

```
    super({private: true}, ['32113123123', '524214691'])
    this.name = 'Account Model'
    this.url +='/accounts/'
  }
  get accountsData() {  ◄——  Get-метод вычисляемого атрибута
    // ... Запрос XHR
    return this.data
  }
}
```

Как же использовать эту абракадабру? Легко:

```
let accounts = new AccountModel(5)
accounts.getName()
console.log('Data is %s', accounts.accountsData)
```

Будет выведен следующий результат:

```
Class name: Account Model
Data is %s 32113123123,524214691
```

Модули

Как вы уже знаете, до ES6 в JavaScript не было поддержки собственных модулей. Разработчики использовали AMD, RequireJS, CommonJS и другие обходные решения. Теперь появились модули с операндами `import` и `export`.

В ES5 теги `<script>` используются с немедленно вызываемой функцией или библиотекой (такой, как AMD), тогда как в ES6 можно предоставить доступ к классу при помощи `export`. Я постоянно работаю с Node.js, поэтому использую библиотеку CommonJS, которая также поддерживает синтаксис модулей Node.js.

CommonJS элементарно используется в браузере со сборщиком Browserify (http://browserify.org). Допустим, файл ES5 module.js содержит переменную `port` и метод `getAccounts`:

```
module.exports = {
  port: 3000,
  getAccounts: function() {
    ...
  }
}
```

В файле ES5 main.js для этой зависимости включается вызов `require('module')`:

```
var service = require('module.js')
console.log(service.port) // 3000
```

В ES6 используются ключевые слова `export` и `import`. Например, так выглядит библиотека в файле ES6 module.js:

```
export var port = 3000
export function getAccounts(url) {
  ...
}
```

В импортирующем файле ES6 main.js используется синтаксис `import {name} from 'my-module'`:

```
import {port, getAccounts} from 'module'
console.log(port) // 3000
```

Также можно импортировать все в переменную `service` в main.js:

```
import * as service from 'module'
console.log(service.port) // 3000
```

Лично я считаю, что модули ES6 создают путаницу. Да, их синтаксис более выразителен, но модули Node.js вряд ли изменятся в ближайшем будущем. Лучше иметь только один стиль для JavaScript в браузере и на сервере, поэтому я буду придерживаться стиля CommonJS/Node.js. Кроме того, на момент написания книги поддержка модулей ES6 в браузерах была недоступна, поэтому для использования модулей ES6 вам понадобится нечто вроде jspm (http://jspm.io).

За дополнительной информацией и примерами обращайтесь по адресу http://exploringjs.com/es6/ch_modules.html. И в любом случае — пишите модульный код JavaScript!

Использование ES6 с Babel

Если вы хотите использовать ES6 прямо сейчас, включите Babel в свой процесс сборки. Babel более подробно рассматривается в главе 3.

Другие возможности ES6

В ES6 найдется много других заслуживающих внимания возможностей, которые вы, скорее всего, использовать не будете (по крайней мере сейчас). Перечислю их в произвольном порядке:

- Новые методы для математических вычислений, чисел, строк, массивов и объектов.
- Двоичные и восьмеричные числовые типы.
- Опрератор расширения.
- Конструкции `for of` (снова привет CoffeeScript!).
- Уникальный неизменяемый тип данных (Symbols).

- Хвостовая рекурсия.
- Генераторы.
- Новые структуры данных (такие, как `Map` и `Set`).

ECMAScript делает работу более эффективной и сокращает количество ошибок. Его развитие продолжится. Нам приходится постоянно учиться. В этом вам помогут следующие ресурсы:

- Краткая сводка ES6, http://reactquickly.co/resources
- Nicolas Zakas, «Understanding ECMAScript 6» (Leanpub, 2017), https://leanpub.com/understandinges6
- Axel Rauschmayer, «Exploring ES6» (Leanpub, 2017), http://exploringjs.com/es6.html
- Учебный курс ES6, https://node.university/p/es6
- Учебный курс ES7 и ES8, https://node.university/p/es7-es8

Азат Мардан

React быстро. Веб-приложения на React, JSX, Redux и GraphQL

Перевел с английского *Е. Матвеев*

Заведующая редакцией	*Ю. Сергиенко*
Ведущий редактор	*К. Тульцева*
Научный редактор	*М. Устимова*
Литературный редактор	*А. Бульченко*
Художественный редактор	*В. Мостипан*
Корректоры	*С. Беляева, Н. Викторова*
Верстка	*Л. Егорова*

Изготовлено в России. Изготовитель: ООО «Прогресс книга».
Место нахождения и фактический адрес: 194044, Россия, г. Санкт-Петербург,
Б. Сампсониевский пр., д. 29А, пом. 52. Тел.: +78127037373.

Дата изготовления: 01.2019. Наименование: книжная продукция. Срок годности: не ограничен.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12 —
Книги печатные профессиональные, технические и научные.

Импортер в Беларусь: ООО «ПИТЕР М», 220020, РБ, г. Минск, ул. Тимирязева, д. 121/3, к. 214, тел./факс: 208 80 01.

Подписано в печать 18.01.19. Формат 70×100/16. Бумага офсетная. Усл. п. л. 45,150. Тираж 1500. Заказ 0000.