React
В ДЕЙСТВИИ

Марк Тиленс Томас

MANNING

React in Action

MARK TIELENS THOMAS

MANNING
SHELTER ISLAND

Марк Тиленс Томас

React
В ДЕЙСТВИИ

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

2019

ББК 32.988-02-018
УДК 004.738.5
Т56

Томас Марк Тиленс

Т56 React в действии. — СПб.: Питер, 2019. — 368 с.: ил. — (Серия «Для профессионалов»).

ISBN 978-5-4461-0999-9

Книга «React в действии» знакомит фронтенд-разработчиков с фреймворком React и смежными инструментами. Сначала вы познакомитесь с библиотекой React, затем освежите материал о некоторых фундаментальных идеях в данном контексте и узнаете о работе с компонентами. Вы на практике освоите чистый React (без транспиляции, без синтаксических помощников), перейдете от простейших статических компонентов к динамическим и интерактивным.

Во второй половине книги рассмотрены различные способы взаимодействия с React. Вы изучите базовые методы жизненного цикла, научитесь создавать поток данных, формы, а также тестировать приложения. На закуску вас ждет материал об архитектуре React-приложения, взаимодействии с Redux, экскурс в серверный рендеринг и обзор React Native.

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

ББК 32.988-02-018
УДК 004.738.5

ISBN 978-1617293856 англ.
ISBN 978-5-4461-0999-9

Краткое содержание

Оглавление

Часть II. Компоненты и данные в React

Эта книга посвящается моей жене Хейли.
Оставайся навсегда...

Предисловие

Когда я стал изучать и применять на практике JavaScript-библиотеку React, сообщество JavaScript только начинало успокаиваться после периода быстрых инноваций и сбоев (читайте: турбулентностей). Библиотека набирала популярность, и я был в восторге от технологичности React, потому что она предлагала реальные перспективы. Ментальная модель казалась солидной, компоненты упрощали создание пользовательских интерфейсов, API был гибким и выразительным, а весь проект казался именно таким, как надо. Утверждения об удобстве использования API библиотеки также способствовали тому, что у меня сложилось о ней хорошее впечатление.

С тех пор изменилось довольно многое и в то же время немногое. Библиотека React в основном осталась той же самой в смысле ее фундаментальных концепций и API, но сформировался и развился целый комплекс знаний и передовых методик и все больше людей работают с ней. Сформировалась экосистема библиотек с открытым исходным кодом, разработаны смежные технологии. Проходят конференции и встречи сообщества, на которых обсуждается React. В версии 16 основная команда разработчиков переписала внутреннюю архитектуру библиотеки таким образом, чтобы новая и предыдущие версии поддерживали обратную совместимость. Они также проложили путь для множества будущих инноваций. Все эти «изменения без особых изменений» являются признаками того, что я считаю одной из самых сильных сторон React: устойчивого соотношения стабильности и инноваций, которое позволяет адаптировать ее, не усложняя пользователям жизнь.

Все эти, а также другие причины дают возможность библиотеке React удерживать свои позиции и приобретать все большую популярность. Она так или иначе используется во многих крупных корпорациях, бесчисленных стартапах и почти в каждой компании среднего звена. Многие компании, не применяющие React, пытаются переключиться на нее, чтобы модернизировать свои интерфейсные приложения.

Библиотека React не только стала очень популярной в Интернете, но и была портирована на другие платформы. Библиотека React Native — порт React для мо-

бильных платформ — также стала важной новинкой. Все это демонстрирует подход React в духе «однажды научившись, писать где угодно». Это представление библиотеки React как платформы означает, что вы не ограничены ее использованием для разработки браузерных приложений.

Забудем о шумихе вокруг React и сосредоточимся на том, чем эта книга может быть полезна вам. Посыл этой книги я понимаю так: она поможет вам разобраться в библиотеке React и научиться эффективно работать с ней и вы начнете создавать улучшенные пользовательские интерфейсы, пусть и не сразу. Моя цель состоит не в раскручивании модного словечка или подталкивании вас к «волшебным» технологиям. Скорее, я делаю ставку на то, что надежная ментальная модель и глубокое понимание библиотеки React в сочетании с практическими примерами позволят вам создавать невероятные проекты с ее помощью независимо от того, будете ли вы делать это самостоятельно или в сотрудничестве с другими разработчиками.

Благодарности

> Не стоит ждать, пока все станет идеальным, и только потом делиться результатами с остальными. Показывайте то, что у вас получилось, как можно раньше и как можно чаще. Промежуточные результаты, в отличие от конечного, никогда не бывают красивыми.
>
> *Кэтмелл Э. Корпорация гениев. Как управлять командой творческих людей*

Немногие амбициозные проекты одиночек становятся результативными. Часто успех целиком приписывают одному человеку или горстке людей, но такая избирательность дает неверное представление о вкладе в работу большой группы участников проекта. Те, кто утверждает, что сделали все самостоятельно, часто не осознают, что им помогли другие пользователи — собственным примером или консультацией. Более того, неспособность использовать преимущество работы в команде снижает шансы на успех и достижение совершенства. Работа в одиночку означает, что вы и только вы можете выполнить задачу, да и то с ограничениями. Сотрудничество — это путь к совершенству, оно обеспечивает взаимосвязь, показывает перспективы, дает новые идеи и бесценную обратную связь.

Я не так глуп, чтобы говорить, что написал эту книгу сам. Мои пальцы нажимали на клавиши, и мое имя указано на обложке, но это не значит, что все было сделано одним человеком. Нет, эта книга, как и все в моей жизни, за что я благодарен, — результат работы большого сообщества умных, скромных, отзывчивых людей, которые были терпеливыми, добрыми, а иногда и жесткими по отношению ко мне.

Во-первых, я хотел бы поблагодарить свою жену Хейли. Она — моя радость, мой лучший друг, мой творческий партнер. Ей понадобилось колоссальное терпение,

пока я работал над этой книгой. Поздние вечера, глубокие ночи и бесконечные разговоры о книге. Супруга — блестящий писатель — помогла мне, когда случился творческий застой. Она ободряла меня, когда казалось, что закончить книгу невозможно. Она всегда постоянна — в любви и в молитве. Она мгновенно успокаивала меня, подбадривала, когда я сомневался в себе, и отмечала со мной радостные моменты. Она была невероятной в течение всего процесса, и я не могу дождаться, когда смогу ответить ей поддержкой и помочь с множеством книг, которые она напишет в будущем. Я безмерно благодарен Хейли.

Хотелось бы поблагодарить и других близких людей за поддержку. У меня невероятная семья. Мать и отец, Эннмари и Митчелл, обнадеживали меня во время написания книги (и в ходе всей моей жизни). Они пообещали прочитать ее от корки до корки, хотя я не буду заставлять их. Братья, Давид и Питер, также поддерживали и поощряли меня. Однако они не давали обещаний, как родители, поэтому я буду читать им ее вслух весь следующий год (или сколько потребуется). Друзья и коллеги также принесли большую пользу, спрашивая: «Уже закончил» — чтобы подтолкнуть меня, и терпели, когда я объяснял принципы работы библиотеки React. Хочу поблагодарить и моих наставников, особенно доктора Диану Павлак Глиэр (Diana Pavlac Glyer), за то, что научили меня думать в верном направлении и записывать свои мысли.

Сотрудники издательства Manning тоже очень помогли мне. Особенная благодарность Марине Майклз (Marina Michaels) — редактору-консультанту по аудитории, Ники Брукнер (Nickie Bruckner) — техническому редактору-консультанту и Герману Фриджерио (German Frigerio) — корректору. Они потратили множество часов на чтение моей книги и помогали мне писать. Без них ее не было бы. Я также хотел бы сказать спасибо Брайану Сойеру (Brian Sawyer) — именно он предложил мне написать эту книгу, и Марьяне Бэйс (Marjan Bace) — за то, что дала такую возможность. Каждый сотрудник издательства Manning стремится помогать людям эффективно получать важные навыки и познавать концепции. Я точно знаю это и буду очень рад и в дальнейшем способствовать образовательной миссии Manning.

О книге

Эта книга о React — JavaScript-библиотеке, предназначенной для разработки пользовательских веб-интерфейсов. Она охватывает и основные концепции API, связанные с созданием React-приложений. По мере чтения книги вы с помощью библиотеки React разработаете демонстрационное социальное приложение. В процессе работы над ним мы рассмотрим различные темы — от добавления динамических данных до рендеринга на сервере.

Аудитория

Эта книга написана для людей, которые хотят изучить React. Не имеет значения, вы программист, вице-президент по техническому проектированию, технический директор, дизайнер, менеджер, студент университета, учащийся образовательного лагеря для программистов или просто пользователь, интересующийся библиотекой React. В зависимости от своих потребностей вы можете сосредоточиться на чтении разных глав. Я расскажу об общих принципах работы библиотеки React в части I книги и буду уточнять концепции и рассматривать более сложные моменты по мере приближения к ее концу.

Идеально, если вы обладаете базовым знанием языка JavaScript. В книге много JavaScript-кода, но это не чистый язык JavaScript. Я не рассматриваю основополагающие понятия этого языка, хотя поверхностно коснусь их, если они имеют отношение к библиотеке React. Вы должны иметь возможность работать с примерами, если знаете основы JavaScript и понимаете, как функционирует в нем асинхронное программирование.

Предполагается, что вы владеете технологическими основами разработки клиентских веб-приложений, пригодится и знание базовых API браузера. Придется иметь

дело с API Fetch, чтобы выполнять сетевые запросы, устанавливать и получать файлы cookie, а также работать с пользовательскими событиями (ввод, щелчки кнопкой мыши и т. д.). Кроме того, вы будете активно взаимодействовать с библиотеками (хотя и не слишком много!). Знание основ современных клиентских приложений поможет извлечь максимальную пользу из этой книги.

К счастью для вас, я упростил текст, исключив из него инструменты и процесс сборки, которые также требуются для создания современных веб-приложений. Исходный код проекта включает все необходимые зависимости и сборочные инструменты, поэтому вам не нужно понимать, например, как работают инструменты Webpack и Babel, чтобы успешно освоить материал. В целом вы должны иметь базовые навыки работы с JavaScript-кодом и некоторыми концепциями клиентских веб-приложений, чтобы получить максимум пользы от чтения этой книги.

Структура издания

Главы книги разделены на три части.

Часть I «Обзор React» знакомит с библиотекой React. В главе 1 рассматриваются основные принципы ее работы. В ней рассказано о ключевых моментах, на которых базируется React, и показано, как они могут вписаться в процесс разработки. Также мы рассмотрим, что библиотека делает, а чего не делает. Глава 2 демонстрирует принципы работы с кодом. Вы изучите API React и создадите простой блок комментариев с помощью компонентов библиотеки.

Часть II «Компоненты и данные в React» — это погружение в React. Вы увидите, как в ней передаются данные (глава 3). В главе 4 изучите жизненный цикл компонентов и приступите к разработке проекта социального приложения. Над этим проектом будете трудиться и в дальнейшем. В главе 4 рассматривается настройка проекта с помощью исходного кода и объясняется, как работать с ним в остальной части книги. Главы 5–9 погружают вас в глубины библиотеки React. Глава 5 охватывает действия с формами и учит работать с данными и их потоками. В главе 6 на основе проделанного в главе 5 создается более сложный компонент React для отображения карт. Главы 7 и 8 касаются маршрутизации — важной составляющей практически любого современного приложения. Вы создадите роутер с нуля и настроите приложение для поддержки нескольких страниц. В главе 8 продолжите работу с маршрутизацией и интегрируете платформу Firebase, чтобы аутентифицировать пользователей. Глава 9 знакомит с тестированием приложений и компонентов React.

Часть III «Архитектура React-приложений» охватывает более сложные темы и фокусируется прежде всего на том, как перевести ваше приложение на использование библиотеки Redux. В главах 10 и 11 представлена библиотека Redux — решение для управления состоянием. После того как приложение перейдет на Redux, в главе 12 поговорим о рендеринге на стороне сервера. В ней рассматривается

также отключение пользовательского роутера для применения React Router. В главе 13 вкратце обсуждается React Native — проект, который позволяет разрабатывать React-приложения на языке JavaScript для мобильных устройств (iOS и Android).

О коде

В книге описаны две основные группы исходного кода. В первых двух главах вы будете работать с кодом вне репозитория проекта. Вы сможете запускать эти примеры кода на сайте Codesandbox.io — веб-сервисе для программистов. На нем код собирается и исполняется в режиме реального времени, поэтому не придется беспокоиться о процессах настройки и сборки.

В главе 4 вы займетесь настройкой исходного кода проекта. Все исходные файлы доступны для загрузки на веб-сайте книги по адресу www.manning.com/books/react-in-action и на сайте GitHub по адресу github.com/react-in-action/letters-social, а окончательный результат проекта опубликован по адресу social.react.sh. Каждая глава или группа разделов имеет собственную ветку Git, поэтому вы можете легко перейти к следующей главе или следовать за развитием проекта на протяжении всей книги. Исходный код опубликован и поддерживается на сайте GitHub, поэтому задавайте возникающие вопросы на GitHub или на форуме книги на странице forums.manning.com/forums/react-in-action.

JavaScript-код приложения должен быть отформатирован с помощью инструмента Prettier (github.com/prettier/prettier) с учетом последней спецификации ECMAScript (ES2017 на момент написания книги). Инструмент Prettier учитывает концепции, синтаксис и методы, изложенные в этой спецификации. Проект включает в себя конфигурацию ESLint, но при желании модифицируйте ее как угодно.

Требования к программному и аппаратному обеспечению

Эта книга не имеет строгих требований к аппаратному обеспечению. Можете использовать любой компьютер — физический или виртуальный (например, на сайте c9.io), но я не буду рассматривать несоответствия, вызванные различиями в средах разработки. Если проблемы возникают во время работы с определенными пакетами, их репозитории, а также сайт Stack Overflow (stackoverflow.com) — лучшие помощники в их решении.

Что касается программного обеспечения, приведу несколько требований и рекомендаций.

- Сборка проекта предусматривает работу с платформой node.js (nodejs.org), поэтому нужно установить ее последнюю стабильную версию (см. главу 4 для получения дополнительной информации о настройке платформы node.js).

- Вам понадобятся текстовый редактор и веб-браузер. Я рекомендую программу Visual Studio Code (code.visualstudio.com), Atom (atom.io) или Sublime Text (www.sublimetext.com).
- На протяжении всей книги вы будете использовать браузер Chrome, а также встроенные в программу инструменты разработчика. Загрузите дистрибутив браузера на странице www.google.com/chrome.

Об авторе

Марк Тиленс Томас (Mark Tielens Thomas) — full-stack-разработчик программного обеспечения и автор. Марк и его супруга живут и работают в Южной Калифорнии. Марк решает проблемы реализации крупных проектов и сотрудничает с ведущими специалистами при разработке высокоэффективных решений. Он религиозен, любит качественный кофе, много читает, обожает быстрые API и отлаженные системы. Пишет книги для издательства Manning и публикует статьи в своем блоге на сайте ifelse.io.

Об обложке

На обложке книги изображен капитан-паша Дерья Бей, адмирал турецкого флота. Капитан-паша был высокопоставленным адмиралом, высшим командиром флота Османской империи. Иллюстрация взята из книги, описывающей коллекцию костюмов жителей Османской империи и опубликованной 1 января 1802 года Уильямом Миллером из «старой» части Бонд-стрит в Лондоне. Обложка той книги утеряна, и мы не смогли узнать ее название. В содержании были приведены подписи рисунков на английском и французском языках, а на каждой иллюстрации указаны имена двух художников, которые работали над ней. Они, безусловно, были бы удивлены, узнав, что их творчество украшает обложку книги по компьютерному программированию... 200 лет спустя.

Книга была куплена сотрудником издательства Manning на антикварном блошином рынке в «Гараже» на 26-й Западной улице на Манхэттене. Продавец приехал из Анкары, и редактор заметил книгу, когда тот уже убирал свой стенд в конце рабочего дня. У редактора не было с собой необходимой суммы наличных денег, а кредитная карта и чек были вежливо отклонены. Вечером продавец улетал обратно в Анкару, поэтому ситуация казалась безнадежной. И каково же было решение? Устное соглашение, скрепленное рукопожатием! Продавец предложил, чтобы деньги были пересланы ему через Интернет, и редактор ушел с листочком

с банковскими реквизитами и книгой под мышкой. Излишне говорить, что на следующий день деньги были перечислены. Мы были благодарны этому неизвестному продавцу и впечатлены его доверием. Вспомнился давно забытый способ заключения устных сделок.

Мы, сотрудники издательства Manning, с большим интересом и удовольствием объединяем компьютерную литературу с выпущенными более двух столетий назад книгами, украшая обложки иллюстрациями, основанными на богатом разнообразии жизненного уклада людей тех времен.

Часть I
Обзор React

Если вы работали над клиентскими JavaScript-приложениями последние пару лет, то, вероятно, слышали о библиотеке React. Возможно, вы слышали о ней, хотя только начинаете создавать пользовательские интерфейсы. И даже если узнаете о библиотеке React из этой книги, я все равно окажусь вам полезен: существует много чрезвычайно популярных приложений, которые разработаны с ее участием. Если вы читаете новости на Facebook, смотрите видео на Netflix или обучаетесь информатике на сайте Khan Academy, то используете приложения, созданные с помощью React.

React – это библиотека для создания пользовательских интерфейсов. Она была разработана в компании Facebook, и с тех пор ее релиз потрясает сообщество почитателей языка JavaScript. За последние несколько лет библиотека приобрела невероятную популярность и является обязательным инструментом для многих команд и разработчиков, создающих динамические пользовательские интерфейсы. Фактически сочетание API React, ментальной модели и надежного сообщества позволило разработать версии библиотеки для других платформ, включая мобильные устройства, и даже виртуальные интерфейсы.

В этой книге мы рассмотрим библиотеку React и разберемся, почему возник такой успешный и полезный проект с открытым исходным кодом. В части I вы начнете с основ библиотеки React и изучите ее с нуля. Инструментарий, связанный с созданием качественных JavaScript-приложений с пользовательским интерфейсом, может быть невероятно сложным. Но чтобы не увязнуть в инструментах, сосредоточимся на изучении всех характеристик React API. Попробуем обойтись без «магии» и постараемся понять, как работает библиотека React.

Из главы 1 вы узнаете основное о библиотеке. Мы рассмотрим некоторые важные идеи, такие как компоненты, виртуальная объектная модель документа и компромиссы, на которые придется пойти при использовании React. В главе 2 я проведу быстрый тур по API React и мы построим простой компонент блока комментариев, чтобы попробовать силы в разработке с помощью React.

1 Что такое React

- Знакомство с React.
- Некоторые концепции и парадигмы React.
- Виртуальная объектная модель документа.
- Компоненты в React.
- React в командной работе.
- Компромиссы использования React.

Если вы работаете веб-разработчиком, то, скорее всего, слышали о React. Возможно, читали публикации в Интернете, например в Twitter или Reddit. Или друг (коллега) упомянул о библиотеке, или вы услышали о ней на конференции. Где бы и как это ни произошло, наверняка то, что вы слышали о React, было одобрительным или немного скептическим. Большинство людей хотят составить собственное мнение о таких технологиях, как React. Эффективные и удачные технологии, как правило, получают положительные отзывы. Зачастую немногие понимают эти технологии до того, как они выходят на более широкую аудиторию. Библиотека React тоже начинала с этого, но теперь пользуется огромной популярностью и широко распространена в мире веб-разработки. И ее известность заслуженна, так как она может многое предложить и позволяет внедрять, обновлять или даже трансформировать ваши идеи пользовательских интерфейсов.

1.1. Знакомство с React

React — это JavaScript-библиотека для создания пользовательских интерфейсов на разных платформах. Она предоставляет мощную ментальную модель для работы и помогает создавать декларативные и ориентированные на компоненты пользовательские интерфейсы. Это то, что библиотека React представляет собой в самом широком и общем смысле. Я раскрою эти идеи и многое другое по ходу книги.

Как библиотека React вписывается в более широкий мир веб-разработки? Часто говорят, что она стоит в одном ряду с такими проектами, как Vue, Preact, Angular, Ember, Webpack, Redux, и другими известными JavaScript-библиотеками

и фреймворками. React часто является важной частью клиентского приложения, и ее функции похожи на присущие библиотекам и фреймворкам, которые я только что упомянул. Фактически сейчас многие популярные интерфейсные технологии относятся к React в большей степени, чем в прошлом. Было время, когда принципы React были новинкой, но с тех пор часть технологий подпала под влияние инновационного подхода, основанного на ее компонентах. Библиотека продолжает поддерживать дух переосмысления установленных лучших практик, главная цель которого — предоставить разработчикам выразительную ментальную модель и эффективную технологию для создания приложений пользовательского интерфейса.

Почему ментальная модель React такая мощная? Она опирается на глубокие области информатики и техники разработки программного обеспечения. Ментальная модель React широко использует функциональные, а также объектно-ориентированные концепции программирования и фокусируется на компонентах как единой основе для разработки. В React-приложениях вы создаете из них интерфейсы. Система рендеринга React управляет ими и автоматически синхронизирует представление приложения. Компоненты часто соответствуют элементам пользовательского интерфейса, например календарям, заголовкам, панелям навигации и т. п., они также могут отвечать за маршрутизацию на стороне клиента, форматирование данных, стилизацию и другие функции клиентского приложения.

Компоненты в React должны быть такими, чтобы их можно было легко понять и интегрировать с другими компонентами. Они должны развиваться в соответствии с предсказуемым жизненным циклом, поддерживать собственное внутреннее состояние и работать со старым добрым JavaScript. В дальнейшем мы изучим эти идеи, а сейчас рассмотрим их вкратце. Обзор основных «ингредиентов», которые входят в React-приложение, представляет рис. 1.1. Кратко рассмотрим их.

- *Компоненты*. Это инкапсулированные блоки функциональности, которые являются основой в React. Они используют данные (*свойства* и *состояние*) для рендеринга пользовательских интерфейсов (мы рассмотрим, как компоненты React работают с данными, в главе 2 и далее). Некоторые их типы также предоставляют набор методов жизненного цикла, которые вы можете перехватывать (hook), иначе — *перехватчики жизненного цикла*. *Процесс рендеринга* (вывод и обновление пользовательского интерфейса на основе ваших данных) в React предсказуем, и компоненты могут подключиться к нему через API React.
- *Библиотеки React*. React содержит набор основных библиотек. Основная библиотека React работает с интерактивными библиотеками `react-dom` и `react-native` и ориентирована на спецификацию и определение компонентов. Она позволяет создавать дерево компонентов, которые можно использовать для средств визуализации (рендеринга) браузера или другой платформы. `react-dom` — одно из таких средств, предназначенное для рендеринга в браузере и на стороне сервера. Библиотеки React Native сфокусированы на нативных платформах и позволяют создавать React-приложения для iOS, Android и других платформ.

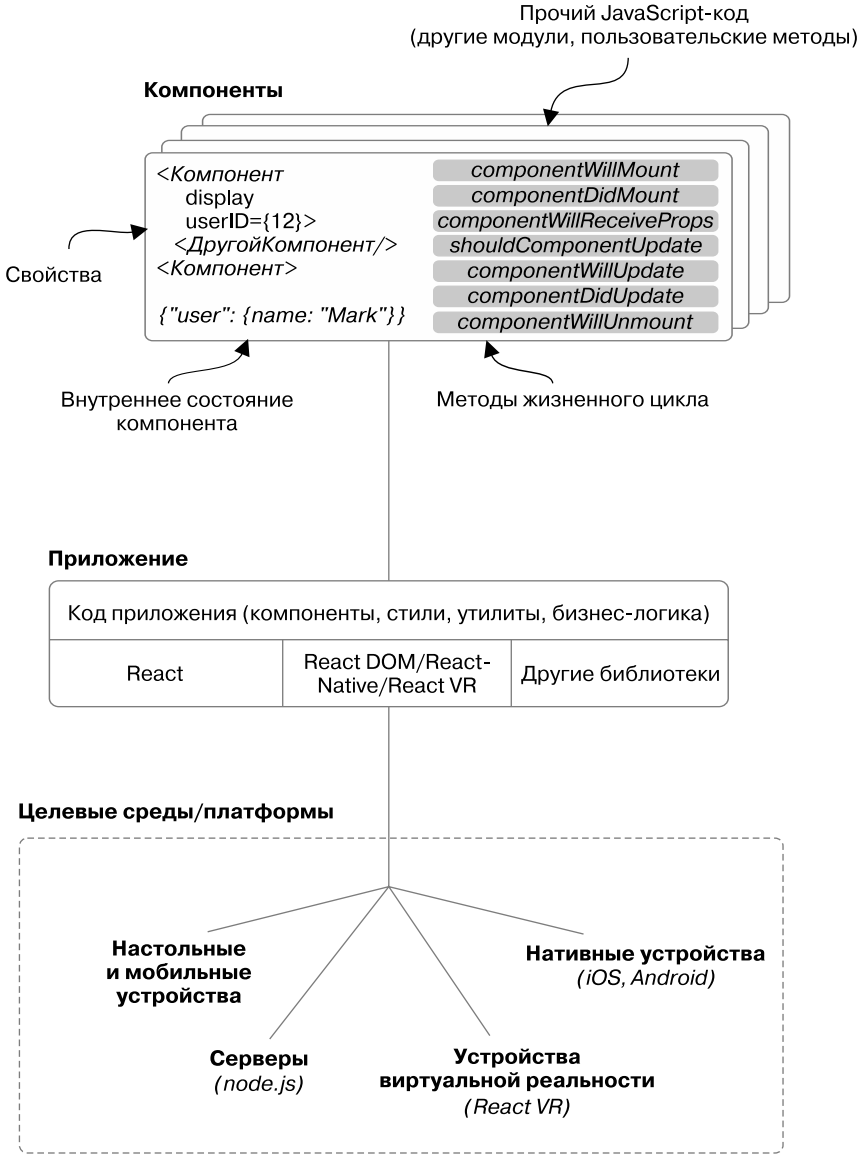


Рис. 1.1. React позволяет создавать пользовательские интерфейсы из компонентов. Последние поддерживают свое собственное состояние, написаны и работают с «ванильным» JavaScript-кодом и наследуют ряд полезных API от React. Большинство React-приложений разработаны для браузеров, но могут использоваться и в таких средах, как iOS и Android. Для получения дополнительной информации о React Native см. книгу React Native in Action, выпущенную издательством Manning

- *Сторонние библиотеки*. React не поставляется с инструментами моделирования данных, HTTP-вызовов, библиотеками стилей или другими распространенными элементами интерфейсного приложения. Поэтому задействуйте в своем приложении дополнительный код, модули или другие инструменты. И хотя этими общими технологиями React не комплектуется, широкая экосистема, окружающая ее, состоит из невероятно полезных библиотек. Далее будем использовать некоторые из них и даже отведем главы 10 и 11 на то, чтобы изучить Redux — библиотеку для управления состоянием.
- *Запуск React-приложения*. React-приложение запускается на платформе, для которой было разработано. В книге мы сосредоточимся на веб-платформе и создадим браузерное и серверное приложения, но другие проекты, такие как React Native и React VR, дают возможность запускать приложения на других платформах.

В этой книге мы потратим много времени на изучение всех закоулков библиотеки React, но перед началом работы у вас может возникнуть несколько вопросов. Подойдет ли вам библиотека? Кто еще применяет ее? Каковы преимущества использования React или отказа от нее? Это лишь часть важных вопросов, на которые следует получить ответы, прежде чем начать изучать React.

1.1.1. Для кого эта книга

Книга предназначена тем, кто разрабатывает пользовательские интерфейсы или планирует начать делать это. А также тем, кто интересуется библиотекой React, но непосредственно не занимается разработкой пользовательских интерфейсов. Вы получите максимальную отдачу от книги, если у вас есть опыт работы с JavaScript и создания интерфейсных приложений с его помощью.

Вы легко научитесь создавать React-приложения, если знакомы с основами языка JavaScript и имеете некоторый опыт разработки веб-приложений. Я не буду освещать основы JavaScript. Такие темы, как прототипная модель наследования, стандарт ES2015 и выше, приведение типов, синтаксис, ключевые слова, паттерны асинхронного программирования, такие как async/await, и другие фундаментальные концепции выходят за рамки этой книги. Я просто рассмотрю все, что особенно важно для разработки React-приложений, но не стану глубоко погружаться в язык программирования JavaScript.

Это не значит, что у вас не получится изучить React или вы ничего не поймете в книге, если не знаете языка JavaScript. Но вы получите гораздо больше, если предварительно уделите время его изучению. Продолжать обучение без знания этого языка может оказаться сложно. Вероятно, вы не во всем сразу разберетесь: код работает, но вы не понимаете как. Это неправильный подход, он не поможет вам как разработчику, поэтому последнее предупреждение: прежде чем изучать React, освойте основы JavaScript. Это замечательный, выразительный и гибкий язык, он вам понравится!

Возможно, вы уже хорошо знаете язык JavaScript и даже имели дело с React. Это не удивляет, учитывая популярность библиотеки. Если это так, вы лучше поймете некоторые основные концепции React. Тем не менее я не буду рассматривать

ряд узких тем, с которыми вы могли столкнуться, работая с библиотекой. Изучить их вы можете, прочитав другие книги издательства Manning, например *React Native in Action*.

Даже если вы не относитесь ни к одной из этих групп, а просто хотите получить краткий обзор библиотеки React, то эта книга и для вас. Вы изучите фундаментальные концепции React и получите доступ к демонстрационному React-приложению. Сможете проверить готовое приложение на странице social.react.sh. Увидите, как основные концепции React-приложения реализуются на практике и как вы или ваша команда можете работать с ними в следующем проекте.

1.1.2. Примечание об инструментарии

Если последние несколько лет вы много работали над клиентскими приложениями, то не удивитесь тому, что инструментарий для приложений стал такой же частью процесса разработки, как и сами фреймворки и библиотеки. Вероятно, вы используете для разработки своих приложений инструментарий типа Webpack, Babel и пр. Как эти и другие инструменты вписываются в эту книгу и что нужно о них знать?

Вам не нужно быть гуру Webpack, Babel и других инструментов, чтобы читать и понимать эту книгу. При создании демонстрационного приложения я применил несколько важных инструментов, и вы сможете просмотреть их конфигурации. Но я не буду подробно описывать их. Инструменты быстро обновляются и изменяются, поэтому их бесполезно глубоко рассматривать в рамках книги. Я обязательно отмечу, если инструменты актуальны или подходят для работы над проектом, но не буду их описывать.

А еще я думаю, что подобный инструментарий способен отвлечь при изучении новых технологий наподобие React. Если вам нужно обдумать несколько новых концепций и парадигм, зачем еще изучать сложный инструментарий? Именно поэтому сначала мы сосредоточимся на изучении «ванильного» React в главе 2, прежде чем переходить к возможностям JSX или JavaScript, реализация которых требует применения инструментов разработки. Но что стоит знать, так это npm. Это инструмент управления пакетами для JavaScript, и мы будем использовать его для установки зависимостей для нашего проекта и выполнения команд для работы с ним в оболочке командной строки. Вероятно, вы уже знакомы с менеджером пакетов npm, но если нет, это не помешает вам читать книгу. Потребуются лишь базовые навыки работы в оболочке командной строки и с менеджером npm. Справочные сведения, касающиеся менеджера пакетов npm, опубликованы на сайте docs.npmjs.com/getting-started/what-is-npm.

1.1.3. Кто использует React

Когда дело доходит до программного обеспечения с открытым исходным кодом, то кто ею пользуется (или не пользуется) — это больше чем просто вопрос популярности. На ответ влияет опыт работы с технологией (включая наличие поддержки, документации, исправлений безопасности), уровень инноваций в проекте и потенциальный жизненный цикл инструментария. Как правило, интереснее, проще

и удобнее работать с инструментами, которые поддерживаются, имеют надежную экосистему и учитывают опыт разработчиков и пользователей.

Библиотека React стартовала как небольшой проект, а теперь имеет широкую популярность и поддержку. Не существует совершенных проектов, и React не исключение, но, поскольку проекты с открытым исходным кодом, как правило, развиваются эффективнее прочих, у нее есть много важных компонентов для достижения успеха. Более того, проект React включает в себя множество других технологий с открытым исходным кодом. Это кажется сложным, поскольку экосистема может стать огромной, но так она формирует надежное и разнообразное сообщество. На рис. 1.2 показана карта экосистемы React. Я буду рассказывать о различных библиотеках и проектах на протяжении всей книги, но если вам интересно узнать об этом больше, вы найдете руководство по адресу ifelse.io/react-ecosystem. Я буду поддерживать и обновлять ресурс по мере развития экосистемы.

Рис. 1.2. Карта экосистемы React разноплановая настолько, что даже не вписывается в мои представления. Если вы хотите получить дополнительную информацию, я опубликовал на сайте ifelse.io/react-ecosystem руководство, которое поможет определить свой путь в экосистеме React

Основной способ прикоснуться к React — это приложения с открытым исходным кодом, которые вы используете ежедневно. Существует много компаний, работающих с React разными интересными способами. Далее перечислены лишь некоторые, применяющие React для своих продуктов: Facebook, Netflix, New Relic, Uber, Wealthfront, Heroku, PayPal, BBC, Microsoft, NFL, Asana, ESPN, Walmart, Venmo, Codecademy, Atlassian, Asana, Airbnb, Khan Academy, FloQast и др.

Эти компании не следуют слепо тенденциям в сообществе JavaScript. У них есть исключительные технические требования, которые влияют на огромное количество пользователей, и им необходимо выполнять разработку в жесткие сроки. Кто-то скажет: «Я слышал, что React — хорошая тема, поэтому мы должны все-таки “реактифицироваться”!» — и не будет прислушиваться к советам менеджеров и других разработчиков. Компаниям и разработчикам нужны хорошие инструменты, которые помогают удачно реализовывать идеи и делать это быстро, создавая высокопроизводительные, масштабируемые и надежные приложения.

1.2. Чего React не делает

До сих пор мы говорили о React в общем — кто ее использует, для кого эта книга и т. д. Мои главные цели при написании книги — научить вас создавать React-приложения и сформировать из вас веб-разработчика. React неидеальна, но с ней по-настоящему приятно работать, и я видел, как команды программистов творят с ее помощью большие дела. Мне нравится писать о React, работать с нею, обсуждать на конференциях, а иногда и вести энергичные дебаты по тому или иному паттерну.

Но я бы оказал вам плохую услугу, если бы умолчал о некоторых недостатках React и не описал, на что библиотека не способна. Понимание того, что что-то нереализуемо, так же важно, как осознание того, чего можно достичь. Почему? Лучшие технические решения обычно результат компромиссов, а не безапелляционного заявления: «React кардинально лучше, чем инструмент X, потому что нравится мне больше». По поводу последнего выражения: вы, вероятно, имеете дело не с двумя совершенно разными технологиями (COBOL против JavaScript). Надеюсь, вы даже не рассматриваете технологии, которые принципиально не подходят для решения этой задачи. И еще: создание больших проектов и решение технических задач никогда не должны зависеть от чьего-то мнения. Дело не в том, что мнения людей не имеют значения (конечно, это не так), просто они не должны влиять на работу.

Компромиссы. Итак, компромиссы необходимы при оценке и обсуждении программного обеспечения. На какие же компромиссы можно пойти относительно React? Во-первых, библиотеку иногда называют *простым представлением*. Это может быть неверно истолковано, если вы подумаете, что React — просто система паттернов типа Handlebars или Pug (née Jade) или что эта библиотека — часть архитектуры MCV (модель — контроллер — представление). Все эти утверждения неверны. Библиотека React не только сочетает оба определения, она — нечто намного большее. Для упрощения опишу React как ответ на вопрос «Что это такое?», а не «Чем она не является?» (например, просто представлением). React — это *декларативная*,

основанная на компонентах библиотека для создания пользовательских интерфейсов, поддерживаемых на различных платформах (Всемирная паутина, нативные устройства, мобильные устройства, серверная платформа, настольные устройства и даже устройства виртуальной реальности в будущем (React VR)).

Это приводит нас к первому компромиссу: React в первую очередь касается особенностей *представления* пользовательского интерфейса. Это означает, что она создавалась не для выполнения многих задач более сложных фреймворка или библиотеки. Сравните с технологией типа Angular. В своем последнем релизе Angular имеет гораздо больше общего с React, чем раньше, с точки зрения концепций и строения, но и предоставляет гораздо больше функций, чем React. Angular включает в себя следующие решения:

- HTTP-вызовы;
- создание и проверку веб-форм;
- маршрутизацию;
- форматирование строк и чисел;
- интернационализацию;
- внедрение зависимостей;
- базовые элементы для моделирования данных;
- настраиваемый фреймворк тестирования (хотя это и не так важно, как другие возможности);
- служебные скрипты (SW) включены по умолчанию (подход с поддержкой обслуживания к выполнению JavaScript-сценариев).

Это очень много, и, судя по моему опыту, пользователи, как правило, реагируют на все эти функции во фреймворке двумя способами. Либо «Ничего себе! Мне не нужны все эти функции!», либо «Ничего себе! Я не могу выбрать, как мне что-то сделать!». Суть таких фреймворков, как Angular, Ember и т. п., состоит в том, что обычно существует четко определенный способ сделать что-то. Например, маршрутизация в Angular реализуется с помощью встроенного роутера Angular Router, все задачи по протоколу HTTP выполняются с помощью встроенных HTTP-процедур и т. д.

Нет ничего принципиально неправильного в этом подходе. Я трудился в таких командах, где использовали подобные технологии, и в таких, где пошли более гибким путем и выбирали технологии, которые хорошо помогают сделать что-то конкретное. Мы отлично поработали с обоими подходами, они хорошо зарекомендовали себя. Я предпочитаю принцип «одно, но качественно», но тут важен компромисс. В библиотеке React нет непосредственных решений для HTTP, маршрутизации, моделирования данных (хотя наверняка есть понятие о потоке данных в ваших представлениях, к которым мы стремимся) или другого функционала, схожего с имеющимся у платформ наподобие Angular. Если ваша команда считает, что в ходе работы над проектом одним фреймворком не обойтись, возможно, React не лучший выбор. Но, по моему опыту, большинство команд одобряют гибкость React, сочетающуюся с ментальной моделью и интуитивно понятными API.

Гибкость React, в частности, заключается в том, что вы можете выбрать наилучшие инструменты для работы. Не нравится, как работает HTTP-библиотека X? Нет проблем, замените ее чем-то другим. Предпочитаете работать с веб-формами иначе? В чем дело, реализуйте свой способ. React предоставляет набор мощных элементов для работы. Справедливости ради стоит сказать: другие фреймворки, типа Angular, как правило, также позволяют заменить что-то, но реально поддерживаемый сообществом способ выполнить поставленную задачу обычно будет встроенным и задействованным.

Очевидный недостаток большей свободы заключается в том, что, если вы привыкли к фреймворкам с более широкими возможностями, например Angular или Ember, вам придется подбирать собственные решения для разных функциональных возможностей своего приложения. Это или хорошо, или плохо в зависимости от таких факторов, как уровень подготовки разработчиков в команде, предпочтения в области управления проектами и другие факторы, характерные для вашей ситуации. Существует много аргументов в пользу как универсального подхода, так и подхода в духе «одно, но качественно». Я склонен придерживаться принципа, который позволяет в каждом конкретном случае принимать гибкие решения таким образом, чтобы команда могла подобрать или создать нужные инструменты. Можно также изучить невероятно большую экосистему JavaScript — вам будет *трудно* найти что-то относящееся к задаче, которую вы решаете. Но факт остается фактом: отличные высокоэффективные команды используют оба подхода (иногда одновременно!), чтобы достичь результата.

Было бы неправильно не упомянуть об особенности, прежде чем рассказывать дальше. Фреймворки JavaScript редко совместимы между собой — это факт. Вряд ли вы создадите приложение, включающее функционал Angular, Ember, Backbone и React, по крайней мере без сегментирования каждой части или жесткого контроля за их взаимодействием. Если можно избежать такой ситуации, нужно так и сделать. Таким образом, разрабатывая приложение, вы обычно работаете с одним или двумя основными фреймворками.

Но что произойдет, если нужно внести в приложение некие изменения? Если вы применяете инструмент с широкими возможностями, например Angular, миграция приложения, скорее всего, приведет к полной переработке кода из-за глубокой идиоматической интеграции фреймворка. Вы можете переписать небольшие части кода приложения, однако, просто заменив несколько функций, не стоит ожидать, что приложение продолжит работать. Вот где проявляется преимущество React. Его использование не означает, что миграция пройдет безболезненно, но вы существенно снизите затраты по сравнению с возникающими при реализации решений с плотно интегрированным фреймворком типа Angular.

Еще один компромисс, на который вы подписываетесь, выбирая React, заключается в том, что библиотека разработана и собрана программистами прежде всего из компании Facebook и предназначена для удовлетворения потребностей пользователей сети Facebook. Вероятно, вам будет нелегко работать с React, если ваше приложение принципиально отличается от того, что нужно этой аудитории. К счастью, большинство современных веб-приложений можно разработать с помощью

React, но, безусловно, есть и такие, для которых она не годится. К ним относятся приложения, которые работают не в традиционных парадигмах пользовательского интерфейса современных веб-приложений или имеют весьма специфические требования к производительности (например, высокоскоростной биржевой тикер). Даже эти проблемы часто решаются с помощью React, но есть ситуации, требующие более специфических технологий.

Последний компромисс, который мы должны обсудить, — это реализация и строение React. Ядро React — это системы, автоматически обрабатывающие пользовательский интерфейс, когда изменяются данные в его компонентах. Они вносят изменения, которые могут перехватить, используя так называемые *методы жизненного цикла*. Мы подробно рассмотрим их в последующих главах. Системы React, поддерживающие обновление пользовательского интерфейса, значительно упрощают создание надежных модульных компонентов, применяемых приложением. То, как библиотека React выполняет бóльшую часть работы по поддержке пользовательского интерфейса с обновлением данных, — еще одна причина того, что разработчикам нравится работать с React и этот мощный инструмент — в ваших руках. Тем не менее не следует полагать, что у «движков» данной технологии нет недостатков или они не используют компромиссы.

React — это абстракция, что чревато затратами. Создаваемая вами система непрозрачна, потому что построена особым образом и управляется через API. Это означает также, что вам нужно разрабатывать пользовательские интерфейсы в режиме, присущем React. К счастью, API React создают «обусловленные выходы», которые позволяют вам спуститься на более низкие уровни абстракции. Вы все равно можете использовать другие инструменты, такие как jQuery, но делать это стоит с оглядкой на React. Это опять-таки компромисс: упрощенная ментальная модель в обмен на то, что у вас не получится делать абсолютно все, что хочется.

Вы не только теряете видимость в базовой системе, но и доверяете действиям React. Это может повлиять на более узкий срез вашего стека приложений (только представления вместо данных, специальные системы генерации веб-форм, моделирование данных и т. д.) и тем не менее имеет значение. Я надеюсь, что вы оцените преимущества React и увидите, что они существенно превышают стоимость обучения и что компромиссы, на которые вы идете при ее использовании, в целом обеспечивают вам гораздо лучшие позиции разработчика. Но я был бы неискренним, если бы сделал вид, что React волшебным образом решит все ваши технические задачи.

1.3. Виртуальная объектная модель документа в React

Мы вкратце поговорили о некоторых возможностях библиотеки React. Я полагаю, что эта информация поможет вам и вашей команде создавать лучшие пользовательские интерфейсы. Частично она связана с ментальной моделью и API, которые предоставляет React. Что скрывается за этим? Основная концепция React — это стремление упростить задачи и исключить ненужную сложность при разработке.

Библиотека React пытается сделать все возможное, чтобы быть исполнителем, освобождая вас для того, чтобы вы могли задуматься о других функциях приложения. Этот подход заключается в том, чтобы позволить вам быть *декларативным*, а не *императивным* разработчиком. Вы объявляете, как компоненты должны вести себя и выглядеть в разных состояниях, а внутренний механизм React справляется со сложностями управления обновлениями, апгрейдом пользовательского интерфейса для отражения изменений и т. д.

Одним из основных управляющих элементов технологии является виртуальная объектная модель документа (*виртуальная DOM*, или *vDOM*). Она представляет собой структуру данных или набор структур данных, имитирующих или отражающих объектную модель документа, которая используется в браузерах. Я говорю «виртуальная объектная модель документа», потому что другие фреймворки, такие как Ember, имеют собственную реализацию подобной технологии. В целом, виртуальная DOM служит промежуточным слоем между кодом приложения и фактической DOM браузера. Как правило, виртуальная DOM позволяет скрыть сложность обнаружения изменений и управления от разработки и перейти к специализированному уровню абстракции. В следующих подразделах мы посмотрим, как этот подход реализуется в React. На рис. 1.3 показана упрощенная схема фактической DOM и ее отношений с виртуальной DOM, которые мы рассмотрим в ближайшее время.

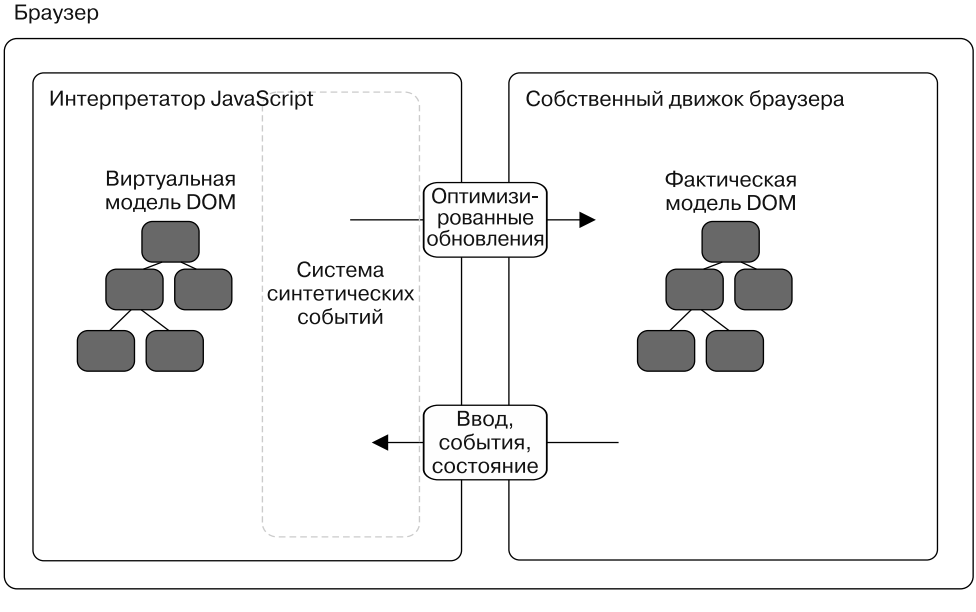


Рис. 1.3. Фактическая и виртуальная DOM. Виртуальная DOM в React обрабатывает изменения в данных, а также преобразование событий браузера в события, которые компоненты могут воспринять и на которые могут среагировать. Виртуальная DOM в React нацелена также на оптимизацию изменений, внесенных в DOM ради производительности

1.3.1. Объектная модель документа

Лучший способ убедиться в том, что вы понимаете, как реализована виртуальная DOM в React, — начать с анализа понимания обычной DOM. Если вы хорошо разбираетесь в DOM, перейдите к следующему подразделу. Если же это не так, начнем с важного вопроса: что такое DOM? DOM, или *Document Object Model (объектная модель документа)*, — это программный интерфейс, который позволяет JavaScript-программам взаимодействовать с различными типами документов (HTML, XML и SVG). Для него существуют стандартизированные спецификации, то есть публичная рабочая группа создала стандартный набор функций, которые должны присутствовать в DOM, и варианты поведения модели. Хотя существуют и другие реализации объектной модели документа, DOM в основном ассоциируется с веб-браузерами, такими как Chrome, Firefox и Edge.

DOM обеспечивает структурированный способ доступа к документу, его хранения и манипулирования различными его частями. В целом DOM представляет собой древовидную структуру, которая отражает иерархию XML-документа. Эта структура складывается из поддеревьев, которые, в свою очередь, состоят из узлов. Вероятно, вы знаете, что это элементы `div` и другие, из которых сформированы ваши веб-страницы и приложения.

Возможно, вы использовали DOM API и раньше, просто могли не знать, что это именно он. Всякий раз, когда вы работаете с методом JavaScript, который обращается к данным, модифицирует или сохраняет данные, связанные с чем-то в HTML-документе, вы почти наверняка применяете DOM или связанные с ней API (для получения дополнительных сведений об интерфейсах веб-API см. сайт developer.mozilla.org/ru/docs/Web/API). Это означает, что не все методы, которые вы задействовали в JavaScript, обязательно являются частью этого языка (`document.findElemenyById`, `querySelectorAll`, `alert` и т. д.). Они входят в большую группу интерфейсов *веб-API* — DOM и других API, которые относятся к браузеру и позволяют взаимодействовать с документами. На рис. 1.4 показана упрощенная версия структуры дерева DOM, которую вы, вероятно, видели на своих веб-страницах.

Общие методы или свойства, которые вы, возможно, задействовали для обновления или запроса веб-страницы, могут включать `getElementById`, `parent.appendChild`, `querySelectorAll`, `innerHTML` и др. Все они обеспечиваются хостовой средой (в данном случае браузером) и позволяют JavaScript взаимодействовать с DOM. Без этого у нас было бы гораздо меньше интересных способов использования веб-приложений и исчезла бы необходимость в книгах о React!

Чаще всего работать с DOM просто, но ситуация может усложниться, если веб-приложение крупное. К счастью, обычно не нужно напрямую взаимодействовать с DOM при создании приложений с помощью React. Вместо этого мы поручаем управление самой библиотеке. Изредка возникают ситуации, когда нужно обойти виртуальную DOM и напрямую общаться с DOM, о чем мы поговорим в следующих главах.

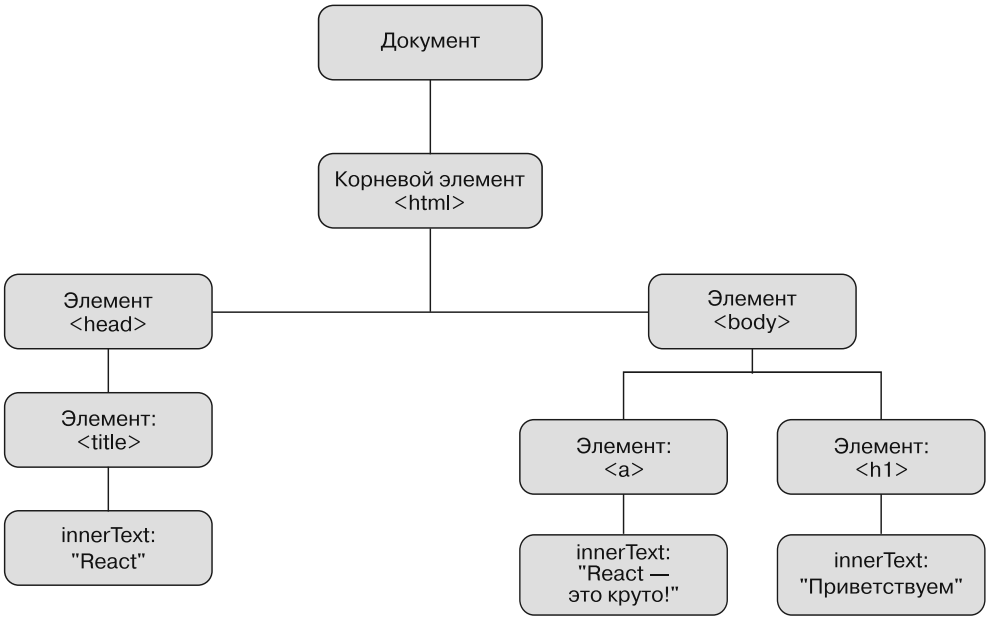


Рис. 1.4. Простая версия структуры дерева DOM, в которой используются элементы, вероятно, вам знакомые. Интерфейс DOM API, который применяется JavaScript, позволяет выполнять операции над этими элементами в дереве

1.3.2. Виртуальная объектная модель документа

Итак, веб-API в браузерах позволяет нам взаимодействовать с веб-документами с помощью JavaScript через DOM. Но если мы уже достигли этого, зачем делать что-то еще? Для начала я хочу отметить, что реализация виртуальной DOM в React не означает, что обычные веб-API плохи или уступают ей. Без них React не может функционировать. Тем не менее в крупных веб-приложениях с DOM возникают кое-какие неприятности. Как правило, они проявляются в области обнаружения изменений. Когда данные изменяются, нужно обновить пользовательский интерфейс, чтобы отразить изменения. Эффективно и просто реализовать это бывает трудно, и React призвана решить эту проблему.

Одна из причин проблемы заключается в том, как браузеры взаимодействуют с DOM. При обращении к DOM-элементу, его модификации или создании браузер часто выполняет запрос через структурированное дерево, чтобы найти нужный элемент. Это просто получение доступа к элементу, которое обычно является только первым этапом обновления. Чаще всего это приводит к изменению компоновки, размеров и выполнению других действий в рамках *изменения*, и все они могут быть

ресурсозатратными. Здесь виртуальная DOM вам не поможет, но поспособствует в оптимизации обновления DOM для учета этих ограничений.

При создании приложения значительного размера, имеющего дело с данными, которые меняются с течением времени, и управлении им часто требуется внести много изменений в DOM. Они нередко конфликтуют или выполняются неоптимальным способом. Это способно чрезмерно усложнить систему, так что разработчикам станет непросто с ней работать, а пользователи могут в ней заблудиться. Таким образом, производительность — еще одно ключевое соображение при построении и реализации React. Использование виртуальной DOM помогает решить эту проблему, и следует отметить, что она разработана довольно быстрой. Надежный API, простая ментальная модель и другие особенности, такие как кросс-браузерная совместимость, в итоге становятся более важными результатами виртуальной DOM в React, чем результат от чрезмерного внимания к производительности. Я об этом говорю, так как вы можете услышать, что виртуальная DOM — это что-то вроде «серебряной пули» для производительности. Да, она производительна, но отнюдь не волшебно, к тому же многие другие преимущества этой модели более важны для работы с React.

1.3.3. Обновления и отличия

Как работает виртуальная DOM в React? Она в чем-то схожа с другим миром программного обеспечения — трехмерными играми. В таких играх иногда используется рендеринг, который работает примерно так: получить информацию с игрового сервера, отправить его в игровой мир (визуальное представление, которое видит пользователь), установить, какие изменения необходимо внести в этот мир, и тогда графическая карта определит необходимый минимум изменений. Одно из преимуществ этого подхода заключается в том, что вам нужны только ресурсы для работы с инкрементальными изменениями, внести которые, как правило, гораздо быстрее, чем если бы пришлось обновлять все.

Таково грубое и поверхностное упрощение способа рендеринга и обновления трехмерных игр. Но это хороший пример, позволяющий понять, как их выполняет React. Плохо реализованное изменение DOM может быть затратным в плане производительности, поэтому React пытается наиболее эффективным способом обновлять пользовательский интерфейс и использует методы, подобные применяемым в трехмерных играх.

Как показано на рис. 1.5, React создает и поддерживает виртуальную DOM в памяти, а средство рендеринга, такое как DOM в React, обрабатывает обновление DOM браузера на основе изменений. React может выполнять интеллектуальные обновления и работать только с измененными частями данных, поскольку способна использовать *эвристическую сверку* для вычисления того, какие части DOM в памяти требуют изменений в фактической DOM. Теоретически это намного более упорядоченно и элегантно, чем грубая сверка или подобные подходы, а основное

следствие для разработчиков — то, что на отслеживание состояния на практике затрачивается меньше сил.

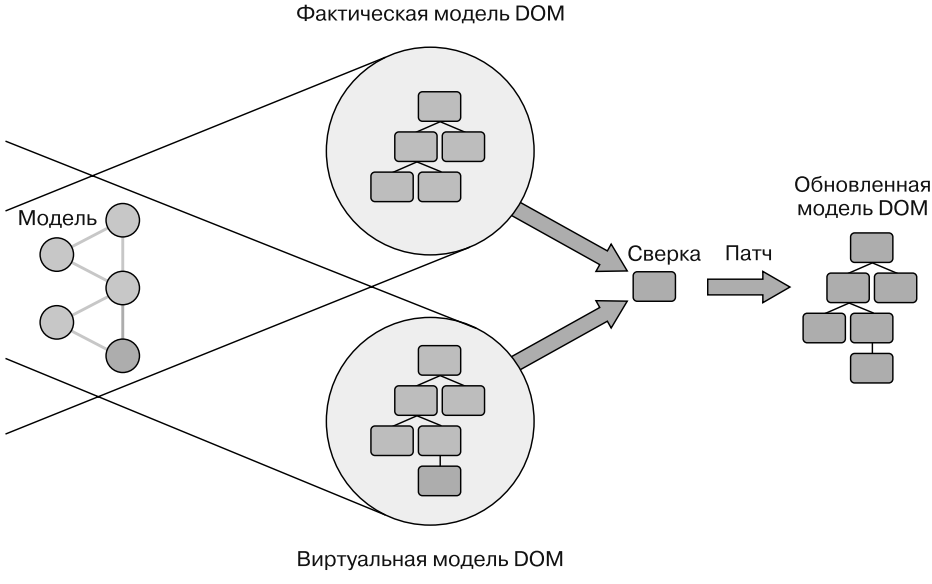


Рис. 1.5. Процедура обновления и учета изменений в React. Когда происходит изменение, React определяет различия между фактическими и внутренними объектами DOM. Затем библиотека эффективно обновляет DOM браузера. Этот процесс часто называют сверкой («что изменилось?») или патчем («обновлять только то, что изменилось»)

1.3.4. Виртуальная DOM: жажда скорости

Как я уже отмечал, виртуальная DOM характеризуется чем-то большим, чем просто скоростью. Она производительна за счет структуры и, как правило, позволяет создавать энергичные и быстрые приложения, которые соответствуют потребностям современных веб-пользователей. Производительность и отличная ментальная модель настолько понравились разработчикам, что многие популярные библиотеки JavaScript создают собственные версии или вариации виртуальной DOM. Даже в этих случаях люди склонны думать, что виртуальная DOM ориентирована в первую очередь на производительность. Производительность — ключевая особенность React, но это второе преимущество после простоты. Виртуальная DOM позволяет не задумываться о сложной логике состояния и сосредоточиться на других, более важных функциях приложения. Скорость и простота улучшают настроение пользователей и разработчиков — это бесспорно!

Мы поговорили о виртуальной DOM, но я не хочу, чтобы вы думали, что это важная часть работы с React. На практике вам не нужно задумываться о том, как

виртуальная DOM обновляет данные или вносит изменения в приложение. В этом и заключается простота библиотеки: у вас появляется время, чтобы сосредоточиться на тех функциях приложения, которые требуют большего внимания.

1.4. Компоненты — базовая единица React

React не просто использует новый подход к работе с изменением данных с течением времени; она также фокусируется на компонентах как на парадигме для организации приложения. Компоненты — основополагающая единица React. Существует несколько разных способов их создания в React, которые мы рассмотрим в следующих главах. Изучить компоненты важно, чтобы понять, как библиотеке React следует работать и как лучше всего применять ее в своих проектах.

1.4.1. Компоненты в целом

Что такое компонент? Если простыми словами, то это часть большего целого. Идея компонентов, скорее всего, вам знакома, и вы, вероятно, часто их видите, даже если этого не понимаете. В качестве как ментального, так и визуального инструмента они позволяют добиться при проектировании и разработке пользовательских интерфейсов улучшенного, интуитивно более понятного структурирования и использования приложений. Компонент может быть тем, чем вы его определите, хотя и не все следует относить к ним. Например, если вы решите, что весь интерфейс — это компонент, без дочерних структур или других компонентов, то примете неправильное решение. Вместо этого полезно разбить разные элементы интерфейса на части, которые можно собрать вместе, многократно применить и легко реорганизовать.

Чтобы разобраться в том, что такое компоненты, рассмотрим образец интерфейса и разделим его на составляющие. На рис. 1.6 показан пример интерфейса, над которым мы будем работать в книге. Пользовательские интерфейсы часто содержат элементы, которые используются многократно или перепрофилируются в других частях интерфейса. И даже если они не применяются повторно, то по крайней мере различаются. Эти отдельные элементы интерфейса можно рассматривать как компоненты. На рис. 1.6, *слева*, показан пример интерфейса, разбитого на компоненты *(справа)*. Мы будем работать над этим интерфейсом позже.

Упражнение 1.1. Разберитесь с компонентами

Посетите популярный сайт, который вам нравится (например, GitHub), и разделите его интерфейс на компоненты. В процессе этого вы, вероятно, обнаружите, что дробите его на все более мелкие части. Когда нужно остановиться? Должно ли отдельное сообщение быть компонентом? Когда имеет смысл использовать мелкие компоненты? Когда целесообразно рассматривать группу элементов как один компонент?

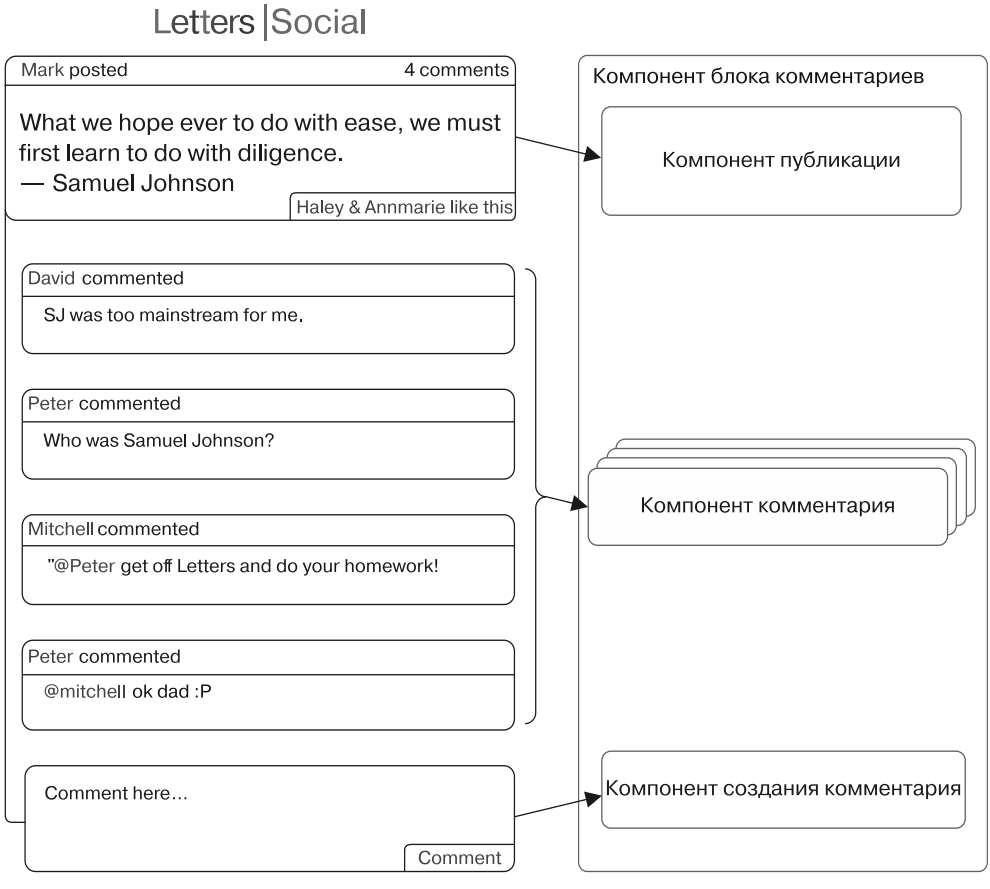


Рис. 1.6. Пример интерфейса, разбитого на компоненты. Каждый отдельный раздел можно рассматривать как компонент, повторяющиеся похожие элементы — как один компонент, который повторно используется для представления разных данных

1.4.2. Компоненты в React: инкапсулированные и многоразовые

Компоненты React хорошо инкапсулированные, многоразовые и составные. Эти характеристики обеспечивают простой и элегантный способ проектирования и создания пользовательских интерфейсов. Ваше приложение может состоять из четких сжатых групп, а не из спагетти-кода. Использование React для создания приложения похоже на сборку модели из деталей LEGO, за исключением того, что деталей точно хватит и нужно единожды создать компоненты, которые вы хотите применять многократно. Вы столкнетесь с ошибками, но, к счастью, здесь нет никаких граблей, на которые реально наступить.

В упражнении 1.1 вы практиковались в определении компонентов и разбивали интерфейс на составные части. Вы могли решить задачу огромным количеством способов и, возможно, были не особенно организованными или последовательными — все в порядке. Но, когда работаете с компонентами в React, важно учесть их организацию и последовательность структурирования. Как правило, вы планируете разработать автономные компоненты и сосредоточиться на определенном элементе, группе элементов или смежных темах.

Это приводит к использованию компонентов переносимых, логически сгруппированных, проще перемещаемых и многократно применяемых в приложении. Хорошо спроектированный компонент React должен быть достаточно автономным, даже если он задействует преимущества других библиотек. Разбиение пользовательского интерфейса на компоненты, как правило, упрощает работу с разными частями приложения. Разграничение компонентов означает, что функциональность и организация могут быть четко определены, а их автономность — что их можно многократно использовать и легко перемещать.

Компоненты в React предназначены для совместной работы. Это означает, что вы можете собирать их вместе для создания новых *составных* компонентов. Формирование структур из компонентов — одна из самых мощных возможностей библиотеки React. Создав компонент один раз, сделайте его доступным для многократного применения в остальной части приложения. Это особенно полезно в крупных приложениях. Если вы работаете в команде среднего или большого размера, публикуйте компоненты в частном реестре (npm и т. п.), где другие команды легко их найдут и используют в новых или существующих проектах. Этот сценарий может не подойти для некоторых команд, но даже более мелкие команды получат выгоду от того, что публикуются компоненты для многократного применения кода.

К последней области использования компонентов в React относятся *методы жизненного цикла*. Это предсказуемые надежные методы, которые вы можете задействовать, если компонент находится на различных стадиях своего жизненного цикла (установка, обновление, размонтирование и т. д.). Мы потратим много времени на изучение этих методов в следующих главах.

1.5. React для командной работы

Вот вы и узнали немного больше о компонентах в целом и о них в React в частности. Библиотека React облегчает вашу жизнь, если вы занимаетесь разработкой самостоятельно. Но как насчет команды? Привлекательные для отдельных разработчиков характеристики React могут заинтересовать и команды специалистов. Как и любая другая технология, это не всегда идеальное решение для какого-то конкретного случая или проекта, независимо от того, что общественное мнение или какие-то разработчики-фанатики попытаются вас убедить в обратном. Как вы

уже знаете, есть области, в которых React не сработает. Но все, что она делает, она делает очень хорошо.

Итак, что превращает React в отличный инструмент для крупных команд и сложных приложений? Во-первых, простота использования. *Простота* — это не то же самое, что *легкость*. Легкие решения часто «грязные», быстрые и, что хуже всего, порождают технический долг. Истинно простая технология гибка и надежна. React реализует это, предоставляя мощные абстракции, с которыми все еще можно работать, обеспечивая погружение в детали нижнего уровня, когда это необходимо. Простую технологию легче понять и применять, потому что была проделана сложная работа по оптимизации и устранению всего лишнего. Библиотека React во многом упростилась, обеспечив эффективное решение без введения вредоносной «черной магии» или непрозрачного API.

Все это отлично подходит для отдельного разработчика, но эффект усиливается, когда работают более крупные команды и организации. Несмотря на то что существует определенная необходимость в улучшении и развитии React, тяжелая работа по созданию простой и гибкой технологии окупается для команд разработчиков. Упрощенные технологии с качественными ментальными моделями, как правило, создают меньшую психологическую нагрузку на разработчиков и позволяют им работать быстрее и эффективнее. В качестве бонуса: более простой набор инструментов проще освоить новым сотрудникам. Ввод нового члена команды в чрезмерно сложный стек не только потребует времени на обучение, но и, вероятно, будет означать, что этот разработчик на протяжении определенного периода не сможет вносить значимый вклад в работу. Поскольку React стремится тщательно пересмотреть сложившиеся лучшие практики, затраты на начальном этапе неизбежны, но результатом станет великая долгосрочная победа.

React — довольно простая библиотека с точки зрения обязанностей и функциональности и этим, безусловно, отличается от аналогов. Если фреймворк типа Angular может потребовать приобретения более сложного API, то React затрагивает только ваше приложение. Это означает, что эту библиотеку гораздо проще интегрировать в текущие проекты, что даст вам время на проработку других характеристик. Некоторые фреймворки и библиотеки придерживаются позиции «все или ничего», а область действия React ограничивается только представлением, и общая совместимость с JavaScript означает, что не всегда все так просто.

Вместо того чтобы придерживаться принципа «все включено», вы можете постепенно переходить к другим проектам или инструментам для React, не меняя радикально структуру, стек или другие связанные области. Это положительная черта любой технологии, и именно так React была впервые опробована в Facebook — в небольшой области проекта. Оттуда она и начала распространяться, так как все больше команд видели и оценивали ее преимущества. Что все это значит для вашей команды? Это значит, что реально задействовать React, не переписывая полностью код проекта.

Простота, ненавязчивый характер и производительность библиотеки React делают ее крайне подходящей как для малых, так и для крупных проектов. По мере изучения React вы увидите, как она может пригодиться вашей команде и использоваться в ваших проектах.

1.6. Резюме

React — это библиотека для создания пользовательских интерфейсов, которая была разработана программистами из компании Facebook и открыта для всех желающих. Это простая и производительная библиотека JavaScript, построенная на основе компонентов. Вместо того чтобы предоставлять полный набор инструментов для создания приложений, React позволяет выбирать, что и как реализовать в моделях данных, серверных вызовах и других функциях приложений. Эти и другие ключевые характеристики библиотеки могут сделать ее отличным инструментом для приложений и команд любого размера. Далее кратко изложены некоторые преимущества React для различных задач.

- *Индивидуальный разработчик*. Изучив React, вы сможете быстро и легко создавать приложения. Крупным командам станет проще обрабатывать ваши приложения, а сложные функции легче реализовать и поддерживать.
- *Разработчик-менеджер*. Разработчикам придется потратить время на изучение React, но в итоге они смогут быстрее и без излишних усилий разрабатывать сложные приложения.
- *Технический директор или высшие руководители*. React, как и любая другая технология, — это рискованные инвестиции. Тем не менее итоговый прирост производительности и уменьшение затрат на человеческие ресурсы часто перевешивают то, что для погружения в тему требуется время. Так можно сказать если не о каждой команде, то точно о многих.

В целом разработчикам несложно выучить React, чтобы в дальнейшем упростить приложение и уменьшить технические затраты за счет многократного использования кода. Потратьте минуту, чтобы повторить то, что вы узнали в этой главе о React.

- React — это библиотека для создания пользовательских интерфейсов, созданная разработчиками из компании Facebook.
- React предоставляет простой, гибкий API, основанный на компонентах.
- Компоненты являются фундаментальной единицей React и широко используются в React-приложениях.
- React реализует виртуальную DOM — прослойку между вашей программой и фактической DOM браузера.

- Виртуальная DOM позволяет эффективно обновлять фактическую DOM с использованием алгоритма быстрой сверки.
- Виртуальная DOM обеспечивает отличную производительность, но самое большое преимущество — это статическая ментальная модель, которую она обеспечивает.

Теперь, когда мы знаем немного больше об истории и строении библиотеки React, можем приступить к работе с ней. В следующей главе создадим свой первый компонент и рассмотрим, как работает React. Вы больше узнаете о виртуальных DOM, компонентах и о том, как создавать собственные компоненты.

2 <Hello world! />: наш первый компонент

- Размышления о пользовательских интерфейсах с компонентами.
- Компоненты в React.
- Как React рендерит компоненты.
- Различные способы создания компонентов в React.
- Использование JSX в React.

В главе 1 были даны теоретические основы React. Если вы жаждете увидеть примеры кода, эта глава для вас. В ней мы подберемся к React поближе. Рассматривая API React, вы создадите простой блок комментариев, который поможет увидеть механику React в действии и освоить ментальную модель работы библиотеки. Мы начнем с создания компонентов React без какого-либо синтаксического сахара или удобств, которые могут скрыть базовую технологию. Поэтому в конце главы рассмотрим JSX (упрощенный язык разметки, который поможет быстрее создавать компоненты React). В следующих главах мы перейдем к более сложным моментам и увидим, как создать из компонентов React полноценное приложение (оно будет называться Letters Social — посмотрите на сайте social.react.sh), а в этой главе ограничимся лишь несколькими связанными компонентами.

Прежде чем погрузиться в детали, вновь бросим беглый взгляд на React, чтобы сориентироваться. На рис. 2.1 приведен обзор основных составляющих большинства React-приложений. Рассмотрим их.

- *Компоненты* — инкапсулированные единицы функциональности, фундаментальные элементы библиотеки React. Это то, из чего сделаны ваши представления. Это функции или классы JavaScript, которые получают свойства как входные данные и поддерживают собственное внутреннее состояние. React предоставляет набор методов жизненного цикла для определенных типов компонентов, поэтому вы можете подключаться к различным этапам управления ими.
- *Библиотеки React* — React-приложения работают, используя библиотеки React. Основную библиотеку React (`react`) поддерживают библиотеки `react-dom` и `react-native`. В React DOM обрабатывает рендеринг в браузере или в серверных средах, тогда как React Native предоставляет собственные привязки, которые позволяют вам создавать React-приложения для iOS или Android.

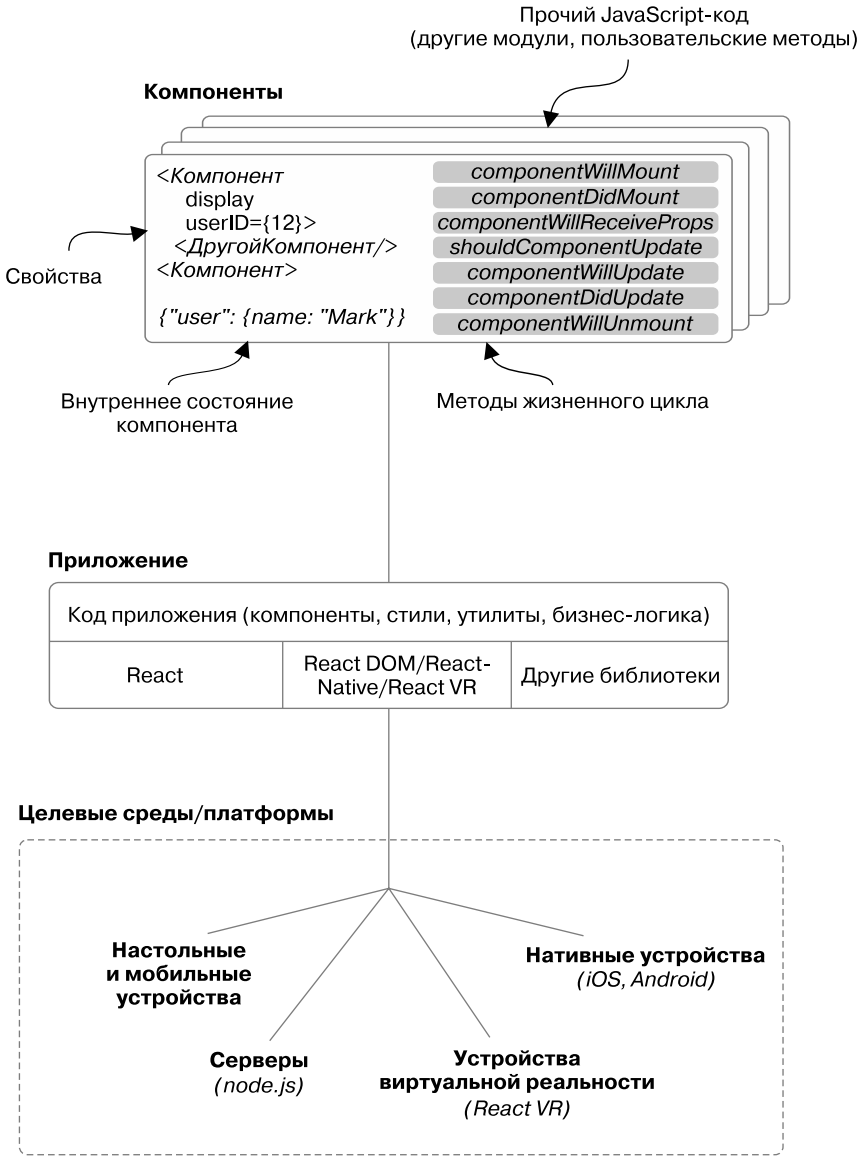


Рис. 2.1. Это очень краткая схема React, которую вы уже видели в главе 1. С помощью React вы можете использовать компоненты для создания пользовательских интерфейсов, которые могут выполняться в браузерах и на нативных платформах, таких как iOS и Android. Это не всеобъемлющий фреймворк — у вас остается свобода выбора библиотек, которые применяются для моделирования данных, стилизации, HTTP-вызовов и т. д. Вы можете запускать React-приложения в браузерах и с помощью React Native на мобильных устройствах

- *Сторонние библиотеки* — React не навязывает собственные подходы к моделированию данных, HTTP-вызовам, конкретным областям стилизации, таким как внешний вид, и другим составляющим вашего приложения. Вы будете интегрировать другие технологии для создания своего приложения по собственному усмотрению. Не все библиотеки совместимы с React, но есть способы интегрировать в нее основную их массу. Мы рассмотрим использование кода, отличного от React, в React-приложениях в главах 4, 10 и 11.
- *Запуск React-приложения* — React-приложение, созданное из компонентов, работает на выбранной вами платформе: веб-сайте, мобильной или нативной.

2.1. Введение в компоненты React

Компоненты — это фундаментальные единицы клиентского приложения, написанного с помощью библиотеки React. Гарантирую, вы будете создавать сотни компонентов! В этой главе вы реализуете простой блок комментариев из них, чтобы быстро познакомиться с React. Но сначала уделим немного времени, чтобы изучить принцип мышления компонентами и посмотреть, как он может влиять на представление блока комментариев. В дальнейшем мы будем погружаться в код, не тратя слишком много времени на планирование, но для первого опыта в React выполним небольшое планирование, чтобы направить мысли в нужное русло. Взгляните на рис. 2.2.

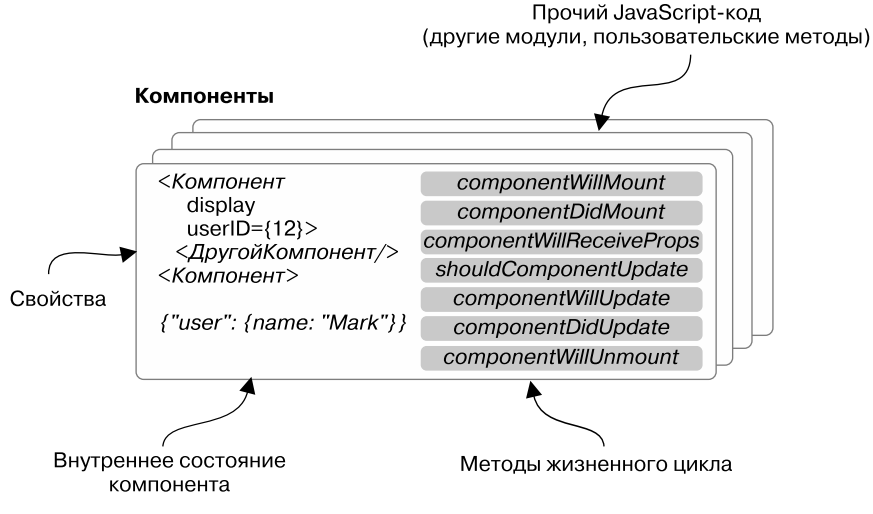


Рис. 2.2. Обзор компонента React. В дальнейшем мы рассмотрим каждую из этих ключевых частей

Предположим, что мы являемся сотрудниками вымышленного стартапа под названием Letters. Вы создадите социальную сеть следующего поколения, где можно

будет размещать информацию, комментировать и ставить «лайки». В этой главе мы исследуем React как потенциальный выбор технологии для вашей компании. Вам было поручено создать простой набор компонентов, чтобы получить представление о технологии. У вас есть лишь грубый набросок, полученный от команды дизайнеров. На рис. 2.3 показана красивая версия того, что вы создадите.

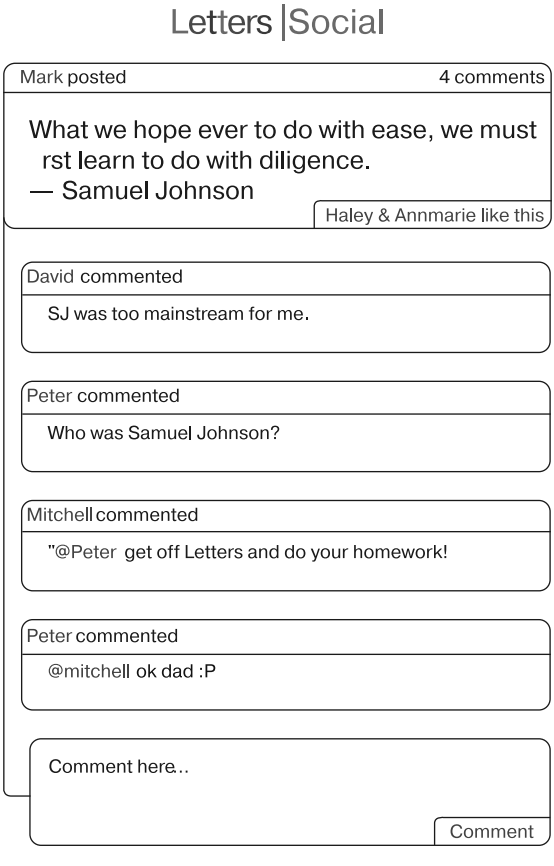
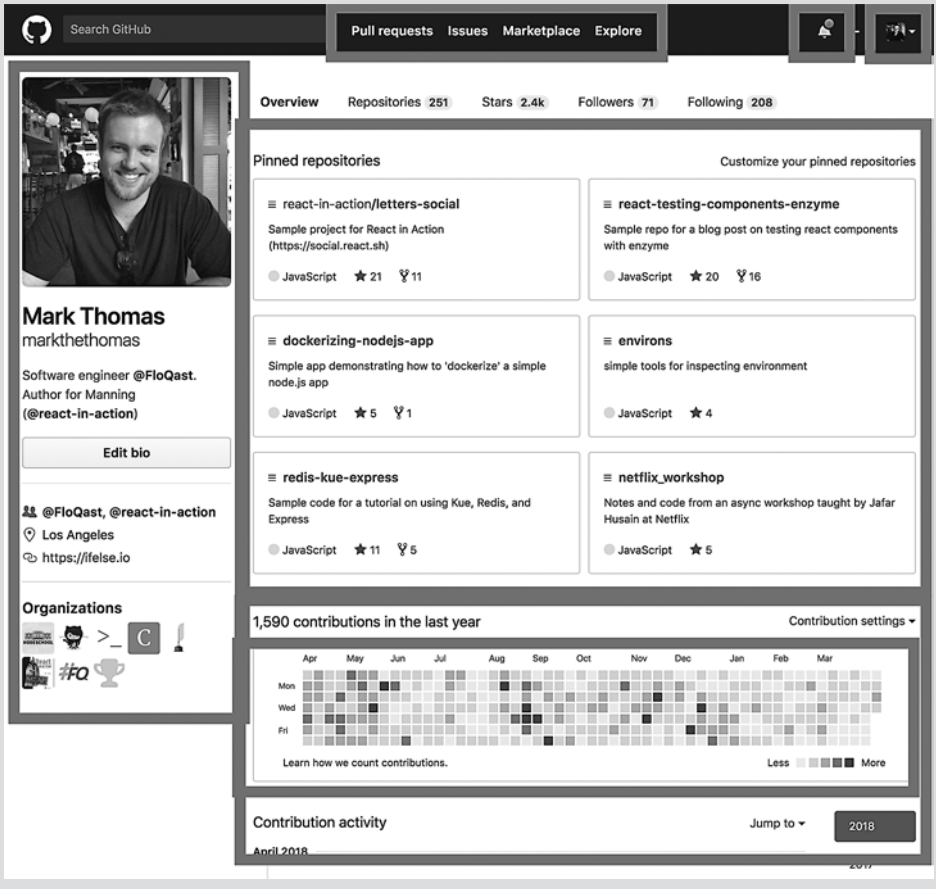


Рис. 2.3. Грубый макет блока комментариев. Вы создадите пользовательский интерфейс, в котором пользователи могут добавлять комментарии к сообщению и просматривать предыдущие комментарии

С чего следует начать? Начнем с изучения данных вашего приложения, а затем посмотрим, как можно превратить эти сведения в компоненты. Как перевести макет в компоненты? Можно было бы просто пытаться создавать компоненты, ничего не зная о React, не понимая, как они работают или какие цели преследуют, и создать что-то путаное или не оптимизированное под React. В нескольких следующих подразделах мы будем заниматься планированием, чтобы получить представление о том, как структурировать и создать блок комментариев.

Упражнение 2.1. Пересмотрите разбивку интерфейса

Прежде чем читать дальше, найдите время, чтобы вернуться к упражнению 1.1 из предыдущей главы. Вы изучили веб-интерфейс и потратили некоторое время, чтобы разметить его самостоятельно. Затратьте еще минутку, чтобы пересмотреть тот же интерфейс и подумать, не разделите ли вы его на компоненты по-другому теперь, когда больше знаете об этих структурах React. Сгруппируете ли что-то иначе? Далее показан размеченный интерфейс страницы профиля на сайте GitHub из главы 1, чтобы освежить вашу память:



2.1.1. Данные приложения

Помимо макета, нам потребуется еще кое-что, прежде чем мы сможем запланировать, как компоненты будут организованы. Нужно знать, какую информацию API станет предоставлять вашему приложению. Глядя на макет, вы, вероятно, уже способны

назвать данные, которые получите. Знакомство с формой данных приложения — важная часть планирования, которое мы выполним, прежде чем создать пользовательский интерфейс.

Веб-API

Возможно, в процессе работы или обучения вы слышали термин *API*. Если вы уже знакомы с данной концепцией, пропустите эту врезку. Если нет, она поможет вам разобраться. Что такое API? API (application programming interface — *интерфейс программирования приложений*) — это набор подпрограмм и протоколов для создания программного обеспечения. Определение может показаться размытым и слишком общим. API — это широкий термин, применимый к множеству явлений — от платформ компании до библиотек с открытым исходным кодом.

В веб-разработке и дизайне API практически стал синонимом удаленного *публичного* API на основе Всемирной паутины. Это означает, что API обычно представляет собой способ взаимодействия с программой или платформой, чаще всего через Интернет. Существует много примеров, два наиболее распространенных — это API Facebook и Stripe. Они предоставляют набор методов для взаимодействия со своими программами и данными через Интернет.

Команда серверных разработчиков из Letters — нашей вымышленной компании — создала для вас такой API. Существует много разных форм и типов веб-интерфейсов API, в этой книге вы будете работать с *RESTful JSON API*. Название означает, что сервер предоставляет данные в формате JSON и доступные данные организованы вокруг таких ресурсов, как пользователи, сообщения, комментарии и т. д. RESTful JSON — распространенные удаленные API, поэтому, вероятно, это не последний раз, когда вы с ним работаете.

В листинге 2.1 приведен пример данных, которые вы получите от API для своего блока комментариев, и показано, как это может соответствовать вашему макету.

Листинг 2.1. Примерный JSON API

```
{
  "id": 123,
  "content": "What we hope ever to do with ease, we must first learn to do
    with diligence. — Samuel Johnson",
  "user": {
    "name": "Mark Thomas",
    "id": 1
  },
  "comments": [{
    "id": 0,
    "user": "David",
    "content": "too. mainstream."
  }, {
    "id": 1,
```

Это не отображается в макете, но это не значит, что вам не нужны эти данные

Вы получили коллекцию объектов комментариев

У комментариев тоже есть идентификаторы

```
    "user": "Peter",
    "content": "Who was Samuel Johnson?"
  }, {
    "id": 2,
    "user": "Mitchell",
    "content": "@Peter get off Letters and do your homework!"
  }, {
    "id": 3,
    "user": "Peter",
    "content": "@mitchell ok dad :P"
  }]
}
```

API возвращает ответ в формате JSON, содержащий отдельную публикацию. У него есть важные свойства, включая `id`, `content`, `author` и `comments`. `id` — число, `content` и `author` — строки, а `comments` — это массив объектов. У каждого комментария есть собственные идентификатор, пользователь, который сделал комментарий, и содержимое комментария.

2.1.2. Несколько компонентов: гибридные и родственные отношения

У вас есть необходимые данные и макет, но как вы собираетесь создавать компоненты для использования этих данных? Во-первых, нужно знать, как их организуют для взаимодействия с другими компонентами. Компоненты React соединены в древовидные структуры. Как и DOM-элементы, они могут быть вложенными и содержать другие компоненты. А еще появляться рядом с другими компонентами, что означает: они встречаются на одном и том же уровне (рис. 2.4).

Напрашивается важный вопрос: в каких отношениях могут состоять компоненты? Вы, вероятно, думаете, что существует довольно много разных типов отношений, создаваемых с применением компонентов, и в некотором смысле вы правы. Способы их использования гибкие. Поскольку они самодостаточны и, как правило, не несут никакого «багажа», их называют *составными*.

Составные компоненты часто легко перемещать и использовать многократно для создания других компонентов. Они напоминают детали LEGO. Каждая такая деталь самодостаточна, поэтому из-за одной детали не нужно переносить целый набор — она легко вписывается в другие комплекты. Переносимость — не главная, но полезная особенность хорошо продуманных компонентов React.

Поскольку компоненты являются составными, их можно применять в разных позициях кода приложения. Где бы они ни использовались, они помогают сформировать определенный тип отношений — *родительский* или *дочерний*. Если один компонент содержит другой, он считается родительским. А тот, который находится внутри другого, — дочерним (потомком). Компоненты, находящиеся на одном уровне, не связаны между собой, хотя могут располагаться рядом друг с другом. Они только «заботятся» о своих родителях и детях.

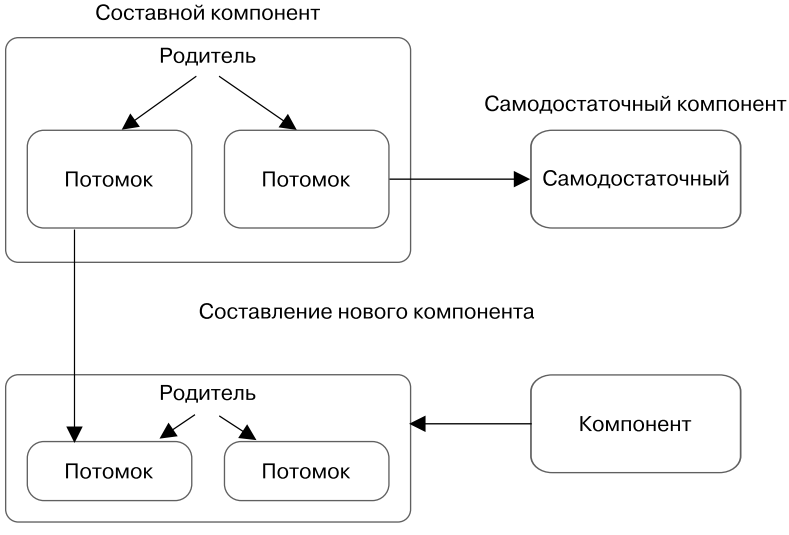


Рис. 2.4. Компоненты могут иметь разные типы отношений (родительский и дочерний), использоваться для создания других компонентов или же быть автономными. Зачастую их можно легко перемещать, потому что они самодостаточны и не тянут за собой «багаж». Они называются составными

На рис. 2.4 показано, как компоненты могут относиться друг к другу в родственном представлении и соединяться для создания новых компонентов. Обратите внимание на отсутствие непосредственных отношений между двумя сестринскими компонентами, несмотря на прямую связь между родителем и потомком. Я остановлюсь на этом подробнее в ходе исследования потока данных в React.

2.1.3. Установление отношений компонентов

Теперь мы знаем, что такое данные и визуальное представление интерфейса, а также компоненты и их родственные отношения. Можете приступить к определению своей иерархии компонентов, то есть применить на практике все то, что изучили. Вы определите, что должно быть компонентом и куда он интегрируется. Процесс установки отношений компонентов различен для каждой команды и каждого проекта. Эти отношения со временем могут измениться, поэтому не рассчитывайте получить совершенную модель с ходу. Простота итераций пользовательского интерфейса — одно из преимуществ React.

Потратьте пару минут, чтобы разделить макет на компоненты, прежде чем читать дальше. Вы уже делали это пару раз, но практика мышления компонентами упростит работу с React. И помните следующее.

- Убедитесь, что компоненты сгруппированы осмысленно: они должны быть организованы в соответствии с функциональностью. Если компоненты в приложении

невозможно передвигать, иерархия может оказаться слишком жесткой. Так происходит не всегда, но нужно отслеживать такие ситуации.

- Если элемент интерфейса повторяется несколько раз, он, как правило, хороший кандидат на то, чтобы стать компонентом.
- Вы не сделаете абсолютно все безупречно в первый раз, и это ожидаемо. Это нормально для итеративного улучшения кода. Первоначальное планирование предназначено не для того, чтобы избежать изменений в дальнейшем, а для определения правильного направления на старте.

Учитывая данные рекомендации, взгляните на доступные данные и макет и начните с разбивки интерфейса на несколько компонентов. На рис. 2.5 показан один из способов сделать это.

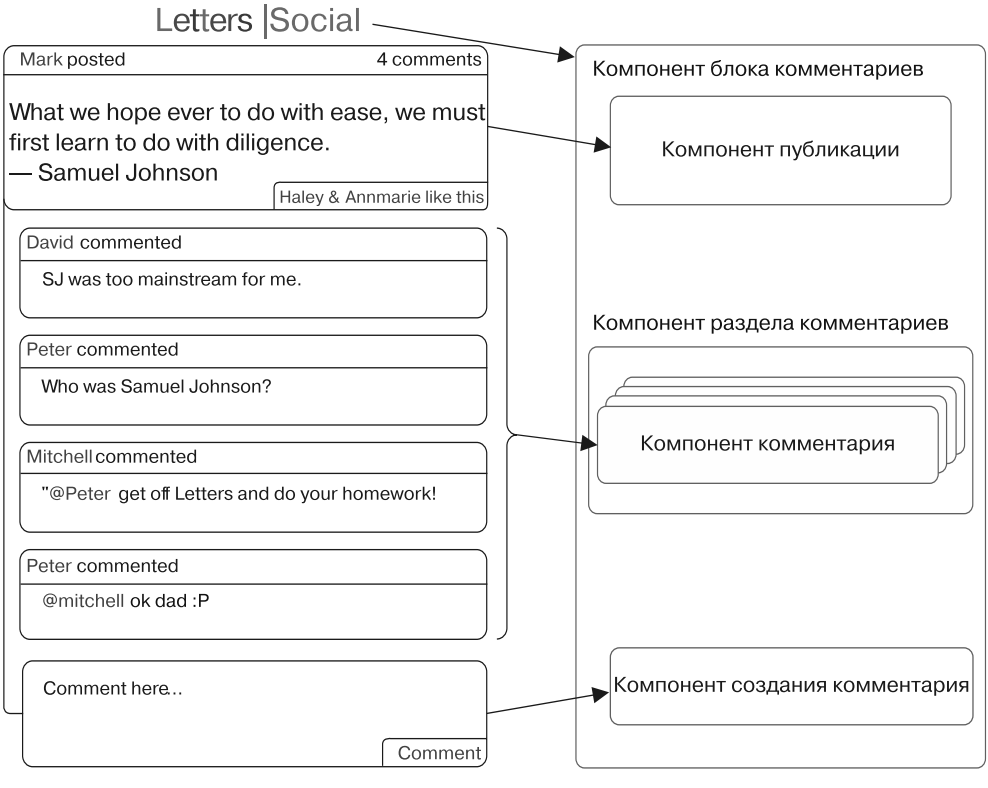


Рис. 2.5. Можете разделить интерфейс на несколько компонентов. Обратите внимание на то, что не обязательно создавать их для каждого элемента интерфейса, хотя имеет смысл превратить больше элементов в компоненты по мере развития приложения. Кроме того, вы заметите, что один и тот же компонент комментария будет использоваться для каждого из комментариев, прикрепленных к публикации. И еще: я нарисовал выноски в стороне для удобства чтения, вы можете сделать их прямо поверх макета

С помощью библиотеки React вы можете гибко подойти к разработке приложения. Я выделил четыре компонента, но существует много и других вариантов. React принудительно устанавливает отношения между родительскими и дочерними компонентами, вдобавок можно свободно определять свою иерархию любым подходящим способом. К примеру, могут возникать ситуации, когда вы разделяете небольшой раздел пользовательского интерфейса на несколько разных частей. Размер пользовательского интерфейса напрямую не зависит от того, сколько компонентов он должен содержать.

Теперь, после выполнения начального планирования, вы готовы создавать пользовательский интерфейс с комментариями. В следующем разделе начнется разработка компонентов React. Вы не будете задействовать синтаксические помощники типа JSX. Вместо этого мы сосредоточимся на сыром коде React, и вы почувствуете мощь технологии, прежде чем переходить к использованию таких помощников.

Вас может разочаровать то, что придется отказаться от ряда помощников, с которыми вы имели бы дело во время обычной работы с React. Я рад этому, потому что вы лучше поймете и оцените выполняемые действия. Так происходит не всегда, но, основываясь на своем опыте, скажу: начиная знакомиться с новой технологией с низкоуровневых элементов, вы, как правило, будете лучше подготовлены к работе с ней. Разумеется, не нужно писать программы JavaScript на ассемблере, но использовать технологию, не до конца понимая ее основы, тоже не очень хорошо.

2.2. Создание компонентов в React

В этом разделе вы создадите несколько компонентов React и запустите их в браузере. Вам не понадобится платформа node.js или что-то подобное, чтобы настроить и запустить их. Вы запустите код в браузере через посредника — сайт CodeSandbox (codesandbox.io). Если предпочитаете редактировать файлы на локальном компьютере, можете нажать кнопку Download (Скачать) в окне редактора кода на сайте CodeSandbox и скачать код для этого примера.

Для своих первых компонентов вы будете использовать три библиотеки: React, React DOM и `prop-types`. React DOM — это средство рендеринга для React, которое было отделено от основной библиотеки React, чтобы лучше разделить задачи. Оно обрабатывает компоненты рендеринга DOM или строку для рендеринга на стороне сервера. `prop-types` — это библиотека разработки, которая поможет выполнить проверку типов данных, переданных вашим компонентам.

Вы разработаете компонент блока комментария, вначале создав его составные части. Так вы лучше поймете, что происходит, когда React создает и отображает ваши компоненты. Вам нужно добавить новый DOM-элемент с идентификатором `root`, а также некоторый базовый код, необходимый React DOM. В листинге 2.2 показано размещение компонента. Для каждого листинга я приведу ссылку на версию кода в Интернете, где можно его редактировать и экспериментировать с ним.

Листинг 2.2. Погнали!

```
//... index.js
const node = document.getElementById("root");

//... index.html
<div id="root"></div>
```

Хранение ссылки на корневой элемент: вы помещаете React-приложение в этот DOM-элемент

В файле index.html вы создали элемент div с идентификатором "root"

Код листинга 2.2 доступен в Интернете по адресу codesandbox.io/s/vj9xkqzkvy.

2.2.1. Создание элементов React

Пока ваш код умеет только загружать библиотеки React и искать DOM-элемент `root`. Чтобы сделать что-то существенное, нужно использовать React DOM. Для создания компонента и управления им необходимо вызвать метод `render` для React. Вы вызовете этот метод с компонентом для рендеринга и элементом `container` (DOM-элемент, который вы ранее сохранили в переменной). Синтаксис `ReactDOM.render` выглядит так:

```
ReactDOM.render(
  ReactElement element,
  DOMElement container,
  [function callback]
) -> ReactComponent
```

React DOM требуются элемент типа `ReactElement` и DOM-элемент. Вы создали допустимый DOM-элемент, который можете использовать, но теперь требуется элемент React. Что такое элемент React?

ОПРЕДЕЛЕНИЕ

Элемент React — легкий неизменяемый примитив в React без изменения состояния. Существует два типа: ReactComponentElement и ReactDOMElement. Элементы ReactDOMElement — это виртуальные представления DOM-элементов. Элементы ReactComponentElement ссылаются либо на функцию, либо на класс, соответствующий компоненту React.

Элементы — это дескрипторы, которые мы используем, чтобы сообщить React, что хотим видеть на экране, они являются центральной концепцией React. Большинство ваших компонентов будут коллекциями элементов React, они создадут «граничное» окружение вокруг части вашего пользовательского интерфейса, чтобы можно было группировать функциональность, разметку и стили. Но что означает фраза «Элемент React является виртуальным представлением DOM-элемента»? То, что элементы React должны относиться к React, как DOM-элементы относятся к DOM — базовым примитивам, из которых состоит пользовательский интерфейс. Когда вы верстаете обычную HTML-разметку, то задействуете различные типы элементов (`div`, `span`, `section`, `p`, `img` и т. д.) для хранения и структурирования информации. В случае с React можно использовать элементы React, которые сообщают

библиотеке о компонентах React или обычных DOM-элементах, которые вы хотите рендерить, для компоновки и создания пользовательского интерфейса.

Возможно, параллель между DOM-элементами и элементами React не сразу станет вам понятна. Это нормально. Помните, как React должна помочь вам, предоставив лучшую ментальную модель для работы? Параллель между DOM-элементами и элементами React — один из способов достижения цели. Это означает, что вы получаете знакомую ментальную структуру — древовидную структуру элементов, похожую на обычные DOM-элементы. Выявить некоторое сходство между элементами React и DOM-элементами поможет рис. 2.6.

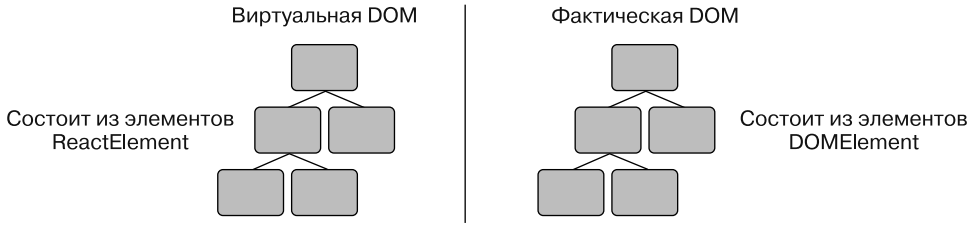


Рис. 2.6. Виртуальная и фактическая DOM имеют сходную древовидную структуру, благодаря чему становится проще понять структуру компонентов и общие принципы React-приложения. DOM состоит из элементов DOMElement (HTMLElement и SVGElement), тогда как виртуальная DOM React — из элементов ReactElement

Можно также представить элементы React как набор основных инструкций для использования React, например, для DOM-элемента. Элементы React — это то, что React DOM будет применять для обновления DOM. На рис. 2.7 элементы React задействуются в общем процессе React-приложения.

Теперь вы знаете немного больше об элементах React в целом, но как они создаются и что происходит в процессе этого? Элементы React создаются с помощью функции `React.createElement`. Взглянем на синтаксис этой функции, чтобы разобраться, как нужно ее использовать:

```
React.createElement(
  String/ReactClass type,
  [object props],
  [children...]
) -> React Element
```

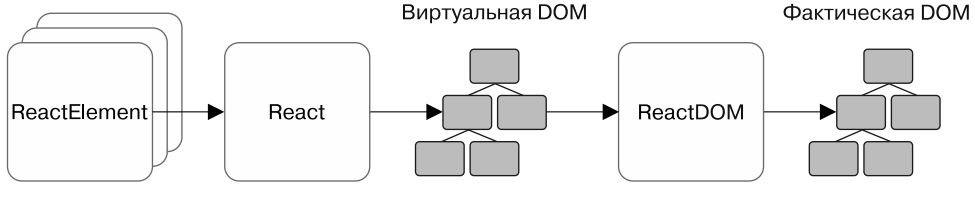


Рис. 2.7. Библиотека React с помощью элементов React создает виртуальную DOM, которой React DOM будет управлять и которую будет использовать для согласования и обновления фактической DOM. Это простые схемы, применяемые React при создании элементов и управлении ими

Функция `React.createElement` принимает строку или компонент (либо класс, расширяющий `React.Component`, либо функцию), объект свойств и дочерние элементы и возвращает элемент React. Помните: элемент React представляет собой простое представление желаемого результата рендеринга средствами React. Он указывает либо на DOM-элемент, либо на другой компонент React.

Подробнее рассмотрим основные инструкции.

- `type` — вы можете передать либо строку, которая является именем тега создаваемого HTML-элемента (`"div"`, `"span"`, `"a"` и т. д.), либо класс React, который мы рассмотрим в ближайшее время. Подумайте об этом в контексте «Что я собираюсь создавать?».
- `props` — краткое название *свойств*. Объект `props` позволяет указать, какие атрибуты будут определены в HTML-элементе (если в контексте `ReactDOMElement`) или будут доступны экземпляру класса компонента.
- `children...` — помните, я говорил, что компоненты React являются составными? Здесь вы можете составить их несколько. Используя строку `children...` и аргументы, переданные после `type` и `props`, можно разместить, упорядочить и даже вложить дополнительные элементы React. Как вы увидите в листинге 2.3, реально встраивать элементы React, вставляя вызовы в `React.createElement` в `children...`.

Функция `React.createElement` задает вопросы «Что я создаю?», «Как это настроить?» и «Что в нем содержится?». Далее показано, как применять функцию `React.createElement`.

Код из листинга 2.3 доступен по адресу codesandbox.io/s/qxx7z86q4w.

Листинг 2.3. Использование функции React.createElement

Отступы отображают вложенные элементы нагляднее. Обратите внимание, как добавлены несколько вызовов функции React.createElement в соответствующих дочерних... параметрах

```
...
    import React, { Component } from 'react';
    import { render } 'react-dom';
    const node = document.getElementById('root');
    const root =
      React.createElement('div', {}, //
        React.createElement('h1', {}, "Hello, world!", //
          React.createElement('a', {href: 'mailto:mark@ifelse.io'},
            React.createElement('h1', {}, "React In Action"),
            React.createElement('em', {}, "...and now it really is!")
          )
        )
      );
    render(root, node); //
...
```

Импорт React и React DOM

Функция React.createElement возвращает один элемент React. Это содержимое root для последующего использования

Внутренний текст также может передаваться в children...

Вызов метода рендеринга, о котором мы говорили ранее

Создание привязки. Обратите внимание, что свойство mailto оформляется так же, как и в обычном HTML-коде

2.2.2. Рендеринг вашего первого компонента

Теперь вы увидите страницу, показанную на рис. 2.8. Только что создан первый компонент React! Используя инструменты разработчика для своего браузера, попробуйте открыть страницу и просмотреть HTML-код. Вы должны увидеть HTML-элементы, соответствующие полученным с помощью React. Обратите внимание на то, что переданные вами свойства также указаны, поэтому можете щелкнуть кнопкой мыши на ссылке и отправить мне электронное письмо с сообщением о том, как вы любите изучать React.

Рис. 2.8. Ваш первый компонент. Результат не особенно впечатляет, но это настоящий компонент React!

Отлично, однако вам может быть интересно, как React превращает функции `React.createElement` в то, что вы видите на экране. Библиотека задействует элементы React, предоставляемые вами для создания виртуальной DOM, которую может использовать `React DOM`, управляя DOM браузера. Помните из рис. 2.4, что виртуальные и фактические DOM имеют сходные структуры? Так и React должна создать собственную виртуальную структуру дерева DOM из ваших элементов React, прежде чем выполнит свою работу.

Для этого React рекурсивно оценивает все свойства `children...` каждого вызова функции `React.createElement`, передавая результат родительскому элементу. Может показаться, что React ведет себя как ребенок, который спрашивает: «Что такое *X*?» — пока не разузнает все об *X*. На рис. 2.9 представлено, как библиотека обрабатывает вложенные элементы React. Идите по стрелкам вниз и вправо, чтобы увидеть, как React исследует `children...` каждого элемента React, пока не сформирует полное дерево.

Теперь, после создания первого компонента, у вас могут возникнуть вопросы и проблемы. Даже отформатировав код, вы поймете, что трудно прочитать компоненты, вложенные всего на несколько уровней. Мы рассмотрим лучшие способы их написания, поэтому не волнуйтесь — вам не придется вставлять вызовы функции `React.createElement` сотни тысяч раз. Применение этого способа в данном примере даст вам представление о том, как работает функция `React.createElement` и, надеюсь, поможет оценить синтаксический сахар JSX, когда вы начнете работать с ним.

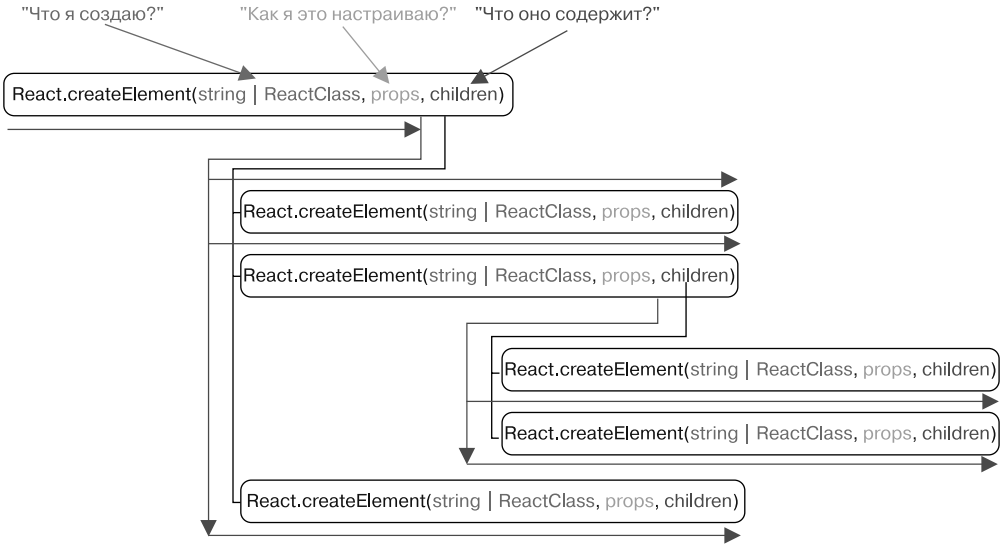


Рис. 2.9. React рекурсивно оценивает ряд элементов React, чтобы определить, как нужно сформировать виртуальную структуру дерева DOM для ваших компонентов. Она также проверит наличие дополнительных элементов React в children… React пройдет всеми возможными путями, как ребенок, спрашивающий: «Что такое X?» — пока не узнает все, что хочет, и не успокоится. Идите по стрелкам вниз и вправо, чтобы понять, как React обрабатывает вложенные элементы React, а также запрашивает каждый параметр

Вам может показаться, что пример слишком прост. До сих пор React кажется такой же, как подробная система шаблонов JavaScript. Но она способна сделать гораздо больше: для этого используются компоненты!

Упражнение 2.2. Элементы React

Прежде чем перейти к компонентам, проверьте, хорошо ли вы понимаете элементы React. Либо на бумаге, либо в уме перечислите несколько характеристик элемента React. Далее напомним некоторые характеристики элементов React.

- Элементы React получают строку, чтобы создать DOM-элемент определенного типа (`div`, `a`, `p` и т. д.).
- Вы можете предоставить конфигурацию элементу React через объект `props`; он аналогичен атрибутам, которые могут иметь DOM-элементы (к примеру, `<img src = "Url-адрес" />`).
- Элементы могут быть вложенными, и можно использовать другие элементы React как потомки каждого элемента.
- Библиотека задействует элементы React для создания виртуальной DOM, которую `React DOM` применяет при обновлении DOM браузера.
- Из элементов React состоят компоненты React.

2.2.3. Создание компонентов React

Как вы, вероятно, уже поняли, использование только элементов React и функции `React.createElement` для создания частей пользовательского интерфейса не дает вам ничего, кроме управления DOM. Вы по-прежнему можете передавать обработчики событий в качестве событий для обработки щелчков кнопкой мыши или ввода текста, передачи других данных для отображения и даже элементов наследования. Но все равно не обойтись без *постоянного состояния*, обеспечиваемого React, методов жизненного цикла, предоставляющих предсказуемые способы работы с компонентом и, если на то пошло, любое логическое группирование, которое способен вам дать компонент. Вам определенно нужен способ группировки элементов React.

Делать это можно с помощью компонентов. Они служат для объединения и группировки функциональных возможностей, разметки, стилей и других связанных элементов вашего интерфейса. Они действуют как своего рода граница вокруг частей вашего пользовательского интерфейса, которые могут содержать и другие компоненты. Компоненты могут быть независимыми, многократно используемыми фрагментами, позволяющими работать с каждой частью по отдельности.

Вы можете создать компоненты двух основных типов с помощью функций и классов JavaScript. В следующих главах я расскажу о первом типе — о функциональных компонентах *без сохранения состояния*. А сейчас поговорим о втором типе — о компонентах React *с сохранением состояния*, созданных с помощью классов JavaScript. Здесь, когда я упоминаю компонент React, я имею в виду, что он создан из класса или функции.

2.2.4. Создание классов React

Чтобы реально создать что-то, нужно больше, чем просто элементы React, — требуются компоненты. Как уже упоминалось, компоненты React (созданные функциями) напоминают элементы React, но имеют больше возможностей. Компоненты в React — это классы, которые помогают группировать элементы и функции React. Они могут быть созданы как классы, расширяющие базовый класс функции `React.Component`. В этом подразделе рассматриваются классы React и идет разговор о том, как их использовать. Посмотрим, как создать класс:

```
class MyReactClassComponent extends Component {
    render() {}
}
```

Вместо того чтобы ссылаться на конкретный метод из библиотеки React, как вы делали с функцией `React.createElement`, для создания компонента с помощью функции `React.Component` объявляется класс JavaScript, который наследуется от абстрактного базового класса `React.Component`. Этому классу обычно необходимо определить метод рендеринга, который будет возвращать один элемент или массив элементов React. Старый способ создания классов React работал с помощью метода `createClass`. С появлением классов в JavaScript все изменилось, и тот способ теперь не используется, хотя вы все еще можете задействовать модуль `create-react-class`, доступный в менеджере npm. Подробные сведения о применении библиотеки React без ES2015 + JavaScript найдете на сайте reactjs.org/docs/react-without-es6.html.

2.2.5. Метод рендеринга

Начнем разбор процесса создания компонентов как классов React с помощью метода `render`, упомянутого ранее. Это один из наиболее распространенных методов, который вы встретите в коде React-приложений, и практически каждый компонент, который отображает что-либо на экране, будет его использовать. В итоге мы рассмотрим компоненты, которые ничего не создают, лишь изменяют или улучшают другие, иногда называемые компонентами *более высокого порядка*.

Метод `render` должен возвращать только один элемент React. Таким образом, этот метод аналогичен тому, с помощью которого создаются элементы React — они могут быть вложенными, но на самом верхнем уровне есть один узел. Однако, в отличие от элементов React, методы `render` классов React имеют доступ к встроенным данным (постоянное внутреннее состояние компонента), а также компонентным методам и дополнительным методам, унаследованным от абстрактного базового класса `React.Component` (обо всех я расскажу). Постоянное состояние, о котором я упоминал, доступно для всего компонента, потому что React создает «экземпляр поддержки» для этого типа компонента. Вот почему вы будете встречать компоненты, которые называются компонентами *с сохранением состояния*.

Все это означает, что React станет создавать и отслеживать специальный объект данных для экземпляра класса React (а не самой схемы создания), который с течением времени сохраняется и может быть обновлен с помощью специальных функций React. Я расскажу об этом подробнее в следующих главах, но на рис. 2.10 показано, что классы React получают экземпляры поддержки, а элементы React — нет.

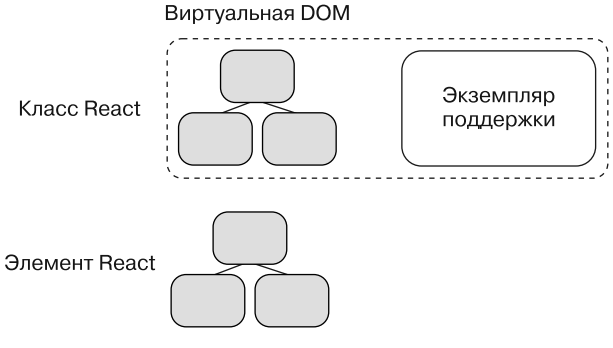


Рис. 2.10. React создаст в памяти экземпляр поддержки для компонентов, созданных как классы компонентов React. Классы компонентов React получают такой экземпляр, а элементы React и компоненты класса, не относящегося к React, — нет. Помните, что элементы React являются зеркалами DOM, а компоненты — способом их группировки. Экземпляр поддержки — это способ хранения данных конкретного компонента и доступа к ним. Данные, хранящиеся в экземпляре, будут доступны для метода рендеринга компонента с помощью определенных методов API. Это означает, что вы получаете доступ к данным, которые можете изменить и которые будут сохраняться с течением времени

При использовании класса React для создания компонента вы получаете также доступ к свойствам, которые можете передать своему компоненту, а он — дочерним компонентам. Можно сохранить эти данные как параметр, который передали

функции `React.createElement`. Как и прежде, его применяют для указания свойств компонентов во время создания. Свойства изменяться не должны, но вскоре вы узнаете способы обновления данных в компонентах React.

В листинге 2.5 в следующем подразделе вы увидите компонент класса React в действии. Будет создано множество вложенных элементов React, и с помощью метода `this.props` будут переданы пользовательские данные. То, что свойство используется с классами React, похоже на создание пользовательского HTML-элемента, такого как `Jedi`, и предоставления ему настраиваемого атрибута, например `name`: `<Jedi name = "Obi Wan"/>`. Я расскажу о ключевом слове `this` в последующих главах, и обратите внимание на то, что в этом случае зарезервированное ключевое слово `this` языка JavaScript указывает на экземпляр компонента.

2.2.6. Проверка свойств с помощью PropTypes

Вы знаете, что компоненты класса React могут задействовать пользовательские свойства, и это замечательно: вы будто можете создавать собственные HTML-элементы, но с расширенной функциональностью. Помните, что с большей мощью приходит бóльшая ответственность. Вам нужен какой-то способ проверять, какие свойства вы будете использовать, чтобы иметь возможность предотвратить ошибки и планировать виды данных, которые станут применять компоненты. Для этого годятся средства проверки (валидаторы), доступные в React в пространстве имен `PropTypes`. Набор валидаторов `PropTypes` входил в основную библиотеку React, но позже был испорчен и помечен как устаревший в версии React 15.5. Чтобы использовать `PropTypes`, необходимо установить пакет `prop-types`, который по-прежнему является частью инструментария React, но уже не включен в основную библиотеку. Этот пакет имеется в исходном коде приложения и примерах на сайте CodeSandbox, с которыми вы работаете в данной главе.

Библиотека `prop-types` предоставляет набор валидаторов, позволяющих указать свойства, в которых нуждается или которых ожидает ваш компонент. Например, вы собираетесь создать компонент ProfilePicture, но это не будет иметь большого смысла без изображения (или логики для обработки, если нет ни одного изображения). Используйте `PropTypes`, чтобы указать, какие свойства должны быть обработаны компонентом ProfilePicture и как они выглядят.

Применение `PropTypes` — это предоставление своего рода контракта, который может быть выполнен или нарушен другими разработчиками или вами в будущем. Для работы React `PropTypes` необязателен, но рекомендован: он позволяет предотвратить ошибки и упростить отладку. Еще одно преимущество работы с `PropTypes` заключается в следующем: если вы укажете, каких свойств ожидаете в первую очередь, то сможете задуматься о том, что вашему компоненту нужно для работы.

Задействуя `PropTypes`, необходимо добавить свойство `propTypes` в класс `React.Component` с помощью свойства статического класса или простым присвоением свойств после определения класса. Обратите внимание на нижний регистр имени свойства класса, в отличие от объекта `React`, так как код легко перепутать. Листинг 2.4 показывает, как используется `PropTypes`, а также возвращаются элементы React из компонентов класса React. В этом листинге вы создаете класс `React`, который можете передать функции `createElement`, добавляете метод `render` и указываете `propTypes`.

Листинг 2.4. Применение PropTypes и метода render

```
import React, { Component } from "react";
import { render } from "react-dom";
import PropTypes from "prop-types";

const node = document.getElementById('root');
class Post extends Component {
    render() {
        return React.createElement(
            'div',
            {
                className: 'post'
            },
            React.createElement(
                'h2',
                {
                    className: 'postAuthor',
                    id: this.props.id
                },
                this.props.user,
                React.createElement(
                    'span',
                    {
                        className: 'postBody'
                    },
                    this.props.content
                )
            )
        );
    }
}

Post.propTypes = {
    user: PropTypes.string.isRequired,
    content: PropTypes.string.isRequired,
    id: PropTypes.number.isRequired
};

const App = React.createElement(Post, {
    id: 1,
    content: ' said: This is a post!',
    user: 'mark'
});

render(App, node);

...
```

Импорт React, React DOM и prop-types

Создание класса React в качестве компонента Post. В этом случае вы указываете propTypes как метод рендеринга

Создание элемента div с классом post

Может сбить с толку в JavaScript, но здесь ссылаемся на экземпляр компонента, а не на сценарий класса React

Использование className вместо имени класса CSS элемента DOM

Опять же свойство content — это внутреннее содержимое создаваемого элемента span

Свойства могут быть опциональными и обязательными, иметь тип. Им даже может потребоваться иметь определенную «форму» (например, объект с конкретными свойствами)

Передаем класс Post React функции React.createElement вместе с некоторыми свойствами для создания элемента. React DOM может это визуализировать — попробуйте изменить данные, чтобы увидеть, как изменяется результат рендеринга вашего компонента

Код листинга 2.4 доступен по адресу codesandbox.io/s/3yj462omrq.

Вы должны увидеть текст: `Mark said: This is a post!`. Если вы не обеспечили определенные свойства, то увидите предупреждение в консоли разработчика. Отсутствие нужных свойств может привести к нарушению работы приложения, так как неизвестно, какие компоненты должны работать, а валидация при этом будет пройдена. Другими словами, если вы забудете предоставить приложению важную

часть данных, вероятно, оно не будет функционировать правильно, а проверка `PropTypes` окажется выполнена и вы получите сообщение, что не указаны свойства. Поскольку `PropTypes` анализирует тип только в режиме разработки, приложение, работающее в обычном режиме, не запустит `PropTypes`.

Теперь, когда вы создали компонент и передали ему данные, попробуйте реализовать вложенные компоненты. Я уже упоминал об этой возможности — одной из многих, делающих React удобной для работы и мощной: компоненты создаются из других компонентов. Листинг 2.5 иллюстрирует этот процесс и демонстрирует специальное использование свойства `children`. Я расскажу об этом подробнее в следующих главах, когда вы начнете изучать маршрутизацию и компоненты более высокого порядка. Применение свойства `this.props.children` сродни выводу вложенных данных. В этом случае вы создаете компонент `Comment`, передаете его как аргумент и получаете его вложение.

Листинг 2.5. Добавление вложенного компонента

```
//...
    this.props.user,
    React.createElement(
        "span",
        {
            className: "postBody"
        },
        this.props.content
    ),
    this.props.children
//...
class Comment extends Component {
    render() {
        return React.createElement(
            'div',
            {
                className: 'comment'
            },
            React.createElement(
                'h2',
                {
                    className: 'commentAuthor'
                },
                this.props.user,
                React.createElement(
                    'span',
                    {
                        className: 'commentContent'
                    },
                    this.props.content
                )
            )
        );
    }
}

Comment.propTypes = {
```

Добавление this.props.children к компоненту Post, чтобы он мог отображать дочерние элементы

Создание компонента Comment аналогично тому, как был создан компонент Post

Объявление propTypes

```
    id: PropTypes.number.isRequired,
    content: PropTypes.string.isRequired,
    user: PropTypes.string.isRequired
};

const App = React.createElement(
    Post,
    {
        id: 1,
        content: ' said: This is a post!',
        user: 'mark'
    },
    React.createElement(Comment, {
        id: 2,
        user: 'bob',
        content: ' commented: wow! how cool!'
    })
);

ReactDOM.render(App, node);
```

Вкладывание компонента Comment в компонент Post

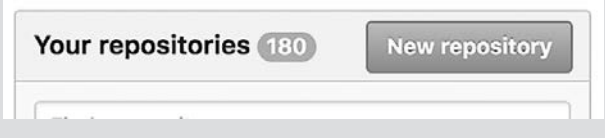
Код из листинга 2.5 доступен по адресу codesandbox.io/s/k2vn448pn3.

Теперь, когда создан вложенный компонент, вы сможете увидеть больше контента в своем браузере.

Далее рассмотрим, как используется упомянутое ранее встроенное состояние, доступное с классами React, для создания динамических компонентов.

Упражнение 2.3. Обратная разработка дерева компонентов

Прежде чем перейти к следующему разделу, проверьте свои знания, занявшись обратной разработкой дерева компонентов с сайта типа GitHub. Откройте панель инструментов разработчика, выберите DOM-элемент, который не слишком глубоко вложен, и реконструируйте класс React. Рассмотрим следующий DOM-элемент.




```
▼<div class="boxed-group flush repos user-repos js-repo-filter" id="your_repos" role="navigation"> == $0
  ▶<div class="boxed-group-action">…</div>
  ▼<h3>
     "Your repositories "
     <span class="counter">180</span>
   </h3>
  ▶<div class="boxed-group-inner">…</div>
   </div>
 </div>
```

Как бы вы сформировали аналогичную структуру компонентов, но в React? (Можете присваивать имена всем классам CSS.)

2.3. Время жизни и время компонента

В этом разделе вы добавите к компонентам Post и Comment код, позволяющий сделать их интерактивными. Ранее мы обнаружили, что компоненты, созданные как классы React, используют специальные способы хранения и доступа к данным через экземпляры поддержки. Чтобы понять это, рассмотрим общие принципы работы React. На рис. 2.11 суммируется то, что вы узнали до сих пор. Можете создавать компоненты из классов React, которые образуются из элементов React (элементы, которые отображаются в DOM). Я называю *классами React* подклассы `React.Component`, с которыми способна работать функция `React.createElement`.

Компоненты, образованные из классов React, имеют экземпляры поддержки, которые позволяют хранить данные и должны обладать методом `render`, возвращающим ровно один React-элемент. Библиотека будет принимать элементы React, создавать из них виртуальную DOM в памяти и поддерживать управление и обновление DOM.

Вы добавили метод `render` и проверку `PropTypes` в классы React. Но для создания динамических компонентов вам понадобится еще кое-что. Классы React могут иметь специальные методы, которые будут вызываться в определенном порядке, так как React управляет виртуальной DOM. Метод `render`, который вы использовали для возврата элементов React, — лишь один из них.

В дополнение к зарезервированным методам жизненного цикла можете добавить собственные методы. React дает вам свободу и гибкость для добавления любых функций, требующихся вашим компонентам. Практически все, что действительно применимо в JavaScript, можно задействовать в React. Если вы взглянете на рис. 1.1, то заметите, что методы жизненного цикла, специальные свойства и пользовательский код составляют бóльшую часть компонента React. Что же еще осталось?

2.3.1. «Реактивное» состояние

Наряду с пользовательскими методами и методами жизненного цикла классы React предоставляют вам состояние (данные), которое может сохраняться вместе с компонентом. В этом помогает экземпляр поддержки, о котором я упоминал. *Состояние* — большая тема, я не буду подробно обсуждать ее в этой главе, но вы узнаете достаточно, чтобы сделать свои компоненты интерактивными и динамическими. Что такое состояние? Определим его как *информацию о чем-то в данный момент*. Вы могли бы, например, узнать состояние друга, спросив: «Как ты себя чувствуешь сегодня?»

Два основных типа состояния — *изменяемое* и *неизменяемое*. Подумайте о различии между ними с точки зрения времени. Может ли состояние измениться после создания? Если да, его называют изменяемым, если нет — неизменяемым. Эти темы рассматриваются в различных научных областях, поэтому я не буду в них углубляться.

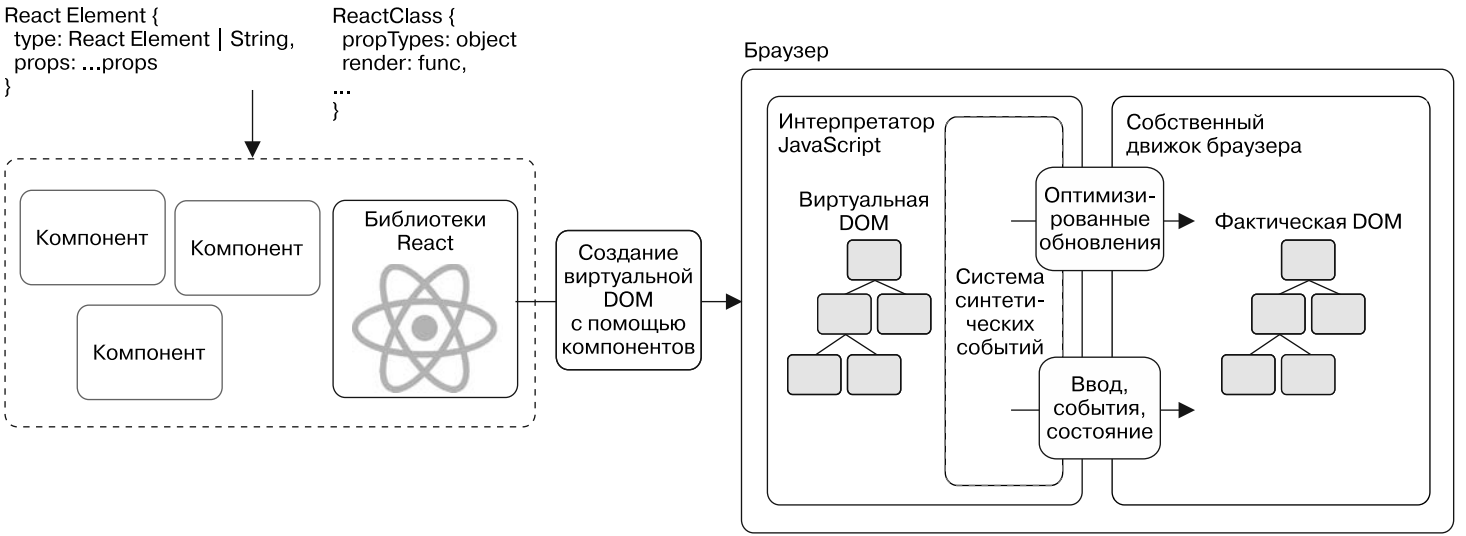


Рис. 2.11. Масштабирование при рендеринге в React. Классы и элементы применяют React для создания в памяти виртуальной DOM, которая управляет фактической DOM. Библиотека создает также «синтетическую» систему событий, чтобы вы могли реагировать на события из браузера, например щелчки кнопкой мыши, прокрутку и другие пользовательские события

В React компоненты, созданные как классы JavaScript (developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Classes), которые расширяют `React.Component`, могут находиться как в изменяемом, так и в неизменяемом состоянии, тогда как компоненты, созданные из функции (функциональные компоненты без состояния), имеют доступ только к неизменяемому состоянию (свойствам — `props`). Я расскажу об этом в следующих главах, а пока буду придерживаться компонентов, которые наследуются от `React.Component` и получают состояние и дополнительные методы. В них состояние доступно из свойства `this.state` экземпляра класса. Неизменяемое состояние доступно с помощью `this.props`, который вы уже использовали для создания статических компонентов.

Свойство `this.props` не следует изменять внутри компонента. Вы увидите способы предоставления данных, которые со временем в них меняются, в следующих главах. На данный момент все, что вам нужно знать, — это то, что нельзя напрямую изменять свойство `this.props`.

Возможно, вам интересно, как работать с `state` и `props` в React. Ответ зависит от того, как вы будете использовать данные, переданные или применяемые в функции. Процесс включает в себя вычисления, отображение, анализ, бизнес-логику и любые другие связанные с данными задачи. Фактически свойства и состояние — это основные способы использования динамических или статических данных в пользовательском интерфейсе (отображение информации пользователя, передача данных обработчикам событий и т. д.).

Состояние и свойства являются механизмами для данных, из которых состоит ваше приложение, и тем, что делает его полезным. Если вы создаете приложение для социальной сети (займемся этим в следующих главах), то часто будете задействовать комбинацию свойств и состояния для создания компонентов, отображающих пользовательскую информацию, обновления и т. д. Если применяете React для рендеринга данных, можете использовать свойства и состояние в качестве ввода для библиотек рендеринга, таких как `D3.js`. Независимо от того, что разрабатывается, вы, вероятно, будете с помощью состояния и свойств управлять информацией в своем приложении React.

Упражнение 2.4. Изменяемое и неизменяемое

Прежде чем читать дальше, проверьте свои знания о различиях между двумя основными типами данных в React — изменяемыми и неизменяемыми. Обозначьте каждое утверждение как истинное или ложное (И | Л).

- Изменяемое состояние означает, что данные могут меняться со временем: И | Л.
- Доступ к состоянию в React получают с помощью свойства `this.state`: И | Л.
- `props` — изменяемый объект, предоставляемый React: И | Л.
- Неизменяемые данные со временем не меняются: И | Л.
- Доступ к свойствам осуществляется через `this.props`: И | Л.

2.3.2. Установка начального состояния

Когда нужно использовать состояние и как начать это делать? Простой ответ таков: *когда вы хотите внести изменения в данные, хранящиеся в компоненте*. Я говорил, что свойства были неизменяемыми (изменить их нельзя), поэтому, если нужно изменить данные, следует изменить состояние. В React данные, которые должны изменяться, зачастую введены пользователем (часто это текст, файлы, выбранные опции и т. д.), но вероятно и множество других вариантов. Чтобы отслеживать взаимодействие пользователей с элементами формы, вам необходимо установить начальное состояние и со временем изменить его. Для установки начального состояния вашего компонента — компонента блока комментариев, который основывается на идеях и концепциях из предыдущих листингов, можете применять конструктор компонента. Это позволит добавлять комментарии к сообщению с помощью простой формы. В листинге 2.6 показано, как настроить компонент и задать начальное состояние.

Листинг 2.6. Установка начального состояния

```
//...
class CreateComment extends Component {
    constructor(props) {
        super(props);
        this.state = {
            content: '',
            user: ''
        };
    }
    render() {
        return React.createElement(
            'form',
            {

                className: 'createComment'
            },
            React.createElement('input', {
                type: 'text',
                placeholder: 'Your name',
                value: this.state.user
            }),
            React.createElement('input', {
                type: 'text',
                placeholder: 'Thoughts?'
            }),
            React.createElement('input', {
                type: 'submit',
                value: 'Post'
            })
        );
    }
}
```

Вызов функции super в конструкторе класса и назначение объекта начального состояния экземпляру свойства состояния класса. Обратите внимание, что вы обычно не назначаете такое состояние, разве что конструктору класса компонента

Создание компонента как класса React, который будет иметь некоторые поля ввода для пользователя. Я расскажу об этом подробнее в следующих главах

```
CreateComment.propTypes = {
    content: React.PropTypes.string
};
//...
const App = React.createElement(
    Post,
    {
        id: 1,
        content: ' said: This is a post!',
        user: 'mark'
    },
    React.createElement(Comment, {
        id: 2,
        user: 'bob',
        content: ' commented: wow! how cool!'
    }),
    React.createElement(CreateComment)
);
```

Добавление CreateComment в компонент приложения

Код листинга 2.6 доступен по адресу codesandbox.io/s/p5r3kwqx5q.

Чтобы обновить состояние, которое вы инициализировали в конструкторе класса компонента, следует использовать специальный метод: вы не можете просто перезаписать `this.state`, как могли бы сделать в случае работы не с React. Так происходит потому, что React должна отслеживать состояние и обеспечивать постоянную синхронизацию виртуальной и фактической DOM. Чтобы обновить состояние в классе компонента React, применяйте метод `this.setState`; синтаксис которого приведен далее. Для обновления состояния требуется функция обновления, которая ничего не возвращает:

```
setState(
  function(prevState, props) -> nextState,
  callback
)-> void
```

Метод `this.setState` принимает функцию обновления, возвращающую объект, который будет частично превращаться в состояние. Например, если вы изначально задали свойство имени пользователя (`username`) как пустую строку, нужно применить метод `this.setState`, чтобы установить новое имя пользователя в качестве состояния вашего компонента. React примет это значение и обновит скрытый экземпляр и DOM с новым значением.

Одним из ключевых различий между обновлением или переназначением значения в JavaScript и использованием метода `setState` является то, что React способна выбирать пакетные обновления на основе изменений состояния, чтобы достичь максимальной эффективности. Это означает, что, когда вы вызываете метод `setState` для обновления состояния, оно не обязательно произойдет сразу. Можно сказать, что React обновит DOM на основе нового состояния самым эффективным способом и максимально быстро.

Что вызывает обновление React? JavaScript управляется событиями, поэтому он, вероятно, будет реагировать на какой-то пользовательский ввод (по крайней мере

в браузере). Это может быть щелчок кнопкой мыши, нажатие клавиши и многие другие события, поддерживаемые браузерами. Как события обрабатываются в React? Она реализует синтетическую систему событий как часть виртуальной DOM, которая будет транслировать события в браузере в события для вашего React-приложения. Вы можете настроить обработчики событий, которые способны реагировать на события из браузера, как обычно в JavaScript. Одно из отличий заключается в том, что обработчики событий React настроены на сами элементы или компоненты React (в отличие от использования слушателя `addEventListener`). Вы можете обновить состояние своего компонента, работая с данными этих событий (введенный текст, значение переключателя или даже цель события).

В листинге 2.7 показано, как реализовать на практике то, что вы узнали о настройке начального состояния и обработчиков событий. В браузере можно прослушивать множество различных событий, охватывающих почти любые взаимодействия с пользователем (щелчок кнопкой мыши, ввод текста, заполнение формы, прокрутка и т. д.). Здесь мы больше всего интересуемся двумя основными событиями: изменением значения ввода формы и ее отправкой. Отслеживая их, вы получаете и используете данные для создания новых комментариев.

Листинг 2.7. Настройка обработчиков событий

```
...
class CreateComment extends Component {
    constructor(props) {
        super(props);
        this.state = {
            content: '',
            user: ''
        };
        this.handleUserChange = this.handleUserChange.bind(this);
        this.handleTextChange = this.handleTextChange.bind(this);
        this.handleSubmit = this.handleSubmit.bind(this);
    }
    handleUserChange(event) {
        const val = event.target.value;
        this.setState(() => ({
            user: val
        }));
    }
    handleTextChange(event) {
        const val = event.target.value;
        this.setState(() => ({
            content: val
        }));
    }
    handleSubmit(event) {
        event.preventDefault();
        this.setState(() => ({
            user: '',
            content: ''
        }));
    }
```

Поскольку компоненты, созданные с помощью классов, автоматически не связываются с методами компонентов, вам нужно связать их в конструкторе

Назначение обработчика события для обработки изменений в поле author — вы получите значение входного элемента event.target.value и используете this.setState для обновления состояния компонента

Создание обработчика событий с аналогичной функциональностью для содержимого комментария

Обработчик событий для отправки формы

Сброс поля ввода после отправки, чтобы пользователь мог отправлять другие комментарии

```
    render() {
        return React.createElement(
            'form',
            {
                className: 'createComment',
                onSubmit: this.handleSubmit
            },
            React.createElement('input', {
                type: 'text',
                placeholder: 'Your name',
                value: this.state.user,
                onChange: this.handleUserChange
            }),
            React.createElement('input', {
                type: 'text',
                placeholder: 'Thoughts?',
                value: this.state.content,
                onChange: this.handleTextChange
            }),
            React.createElement('input', {
                type: 'submit',
                value: 'Post'
            })
        );
    }
}
CreateComment.propTypes = {
    onCommentSubmit: PropTypes.func.isRequired,
    content: PropTypes.string
};
...
```

Код из листинга 2.7 доступен по адресу codesandbox.io/s/x9mxo31pxp.

Вы обратили внимание, как используется метод `.bind` в конструкторе класса компонента? В предыдущих версиях React библиотека *автоматически* привязывала методы к экземпляру вашего компонента. Однако с переходом на классы JavaScript делать это придется самостоятельно. Если вы определили компонентный метод и он не работает, убедитесь, что правильно связали свои методы, — это легко забыть при первом знакомстве с React.

Затем попробуйте не применять обработчики событий `onChange` и посмотрите, получится ли ввести что-либо в поля формы. Ничего не получится, так как React отслеживает синхронизацию фактической DOM с виртуальной, которая не обновляется, и, таким образом, не позволит изменить DOM. Если вы сейчас не понимаете тему до конца, не волнуйтесь — в главах 5 и 6 формы рассматриваются более подробно.

Теперь, когда получен способ прослушивания событий и изменения состояния компонента, можете реализовать способ создания новых комментариев с использованием однонаправленного потока данных. В React потоки данных передаются сверху вниз, от родителей к потомкам (дочерним компонентам). При создании составных компонентов можно передавать информацию дочерним компонентам через свойства

и применять их в дочерних компонентах. Это означает, что вы можете хранить данные из компонента `CreateComment` в родительском компоненте, а затем передавать данные дочерним. Но как получить данные из нового комментария (в том виде, в котором пользователь вводит текст) в дочернем компоненте и передать обратно в родительский и дочерний? На рис. 2.12 показан пример нужного потока данных.

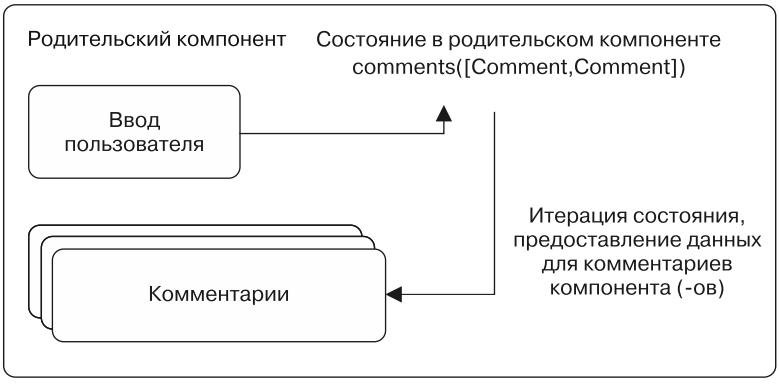


Рис. 2.12. Чтобы добавить сообщение, необходимо захватить данные из полей ввода и каким-то образом отправить их в родительский компонент. Позже обновленные данные будут использоваться для рендеринга сообщений

Как это сделать? Одним из видов данных, которые мы не рассматривали при изучении свойств, являются функции. Поскольку существует возможность передавать функции как аргументы другим функциям в JavaScript, используйте это в своих интересах. Вы можете определить метод для родительского компонента и передать его дочернему в качестве свойства. Таким образом, дочерний компонент может отправить данные обратно своему родителю, не зная, как тот будет обрабатывать данные. Если нужно изменить то, что произошло с данными, с `CreateComment` ничего делать не придется. Чтобы выполнить функцию, переданную в качестве свойства, дочернему компоненту нужно только вызвать метод и передать ему какие-либо данные. В следующем листинге показано, как применять функции в качестве свойств.

Листинг 2.8. Использование функций в качестве свойств

```
//...
class CreateComment extends Component {
    constructor(props) {
        super(props);
        this.state = {
            content: '',
            user: ''
        };
        this.handleUserChange = this.handleUserChange.bind(this);
        this.handleTextChange = this.handleTextChange.bind(this);
        this.handleSubmit = this.handleSubmit.bind(this);
    }
```

```
    handleUserChange(event) {
        this.setState(() => ({
            user: event.target.value
        }));
    }
    handleTextChange(event) {
        this.setState(() => ({
            content: event.target.value
        }));
    }
    handleSubmit(event) {
        event.preventDefault();
        this.props.onCommentSubmit({
            user: this.state.user.trim(),
            content: this.state.content.trim()
        });
        this.setState(() => ({
            user: '',
            text: ''
        }));
    }
    render() {
        return React.createElement(
            'form',
            {
                className: 'createComment',
                onSubmit: this.handleSubmit
            },
            React.createElement('input', {
                type: 'text',
                placeholder: 'Your name',
                value: this.state.user,
                onChange: this.handleUserChange
            }),
            React.createElement('input', {
                type: 'text',
                placeholder: 'Thoughts?',
                value: this.state.content,
                onChange: this.handleTextChange
            }),
            React.createElement('input', {
                type: 'submit',
                value: 'Post'
            })
        );
    }
}

//...
```

Вызов функции onCommentSubmit, которая была передана родителем как свойство, — вы передаете данные из формы и перезагружаете форму, чтобы пользователь знал, что его действие выполнено успешно

Не забудьте связать метод, который вы настроили, с событием onSubmit — без него не будет связи между правильным событием и вашим методом

Код из листинга 2.8 доступен по адресу codesandbox.io/s/p3mk26v3lx.

Теперь, когда ваш компонент способен передать новые данные комментария родителю, нужно добавить определенную разметку, чтобы начать комментирование. В следующих главах вы будете работать с API Fetch API и RESTful JSON API, но использование некоторых элементов разметки пригодится прямо сейчас.

В листинге 2.9 показано, как можно разметить некоторые базовые данные с соответствующими комментариями.

Листинг 2.9. Макетирование данных API

```
...
const data = {
    post: {
        id: 123,
        content:
            'What we hope ever to do with ease, we must first learn to do
            with diligence. — Samuel Johnson',
        user: 'Mark Thomas',
    },
    comments: [
        {
            id: 0,
            user: 'David',
            content: 'such. win.',
        },
        {
            id: 1,
            user: 'Haley',
            content: 'Love it.',
        },
        {
            id: 2,
            user: 'Peter',
            content: 'Who was Samuel Johnson?',
        },
        {
            id: 3,
            user: 'Mitchell',
            content: '@Peter get off Letters
                      and do your homework',
        },
        {
            id: 4,
            user: 'Peter',
            content: '@mitchell ok :P',
        },
    ],
};
...
```

Настройка разметки для компонента CommentBox

Вы будете использовать эти объекты комментариев как имеющиеся комментарии

Затем вам потребуется способ отобразить все комментарии. В React это сделать легко. У вас уже есть компонент, который будет отображать комментарий. Поскольку все, что нужно для работы с компонентами React, — это обычный JavaScript-код, задействуйте функцию `.map()` для возврата нового массива элементов React. Вы не можете применить функцию `.forEach()`, потому что она не возвращает массив и оставит `React.createElement()` без каких-либо данных.

Однако можете сформировать массив, использующий `forEach`, а затем передать данные в него.

Помимо итерации по существующим комментариям, вам необходимо определить метод, который вы передаете компоненту `CreateComment`. Он должен изменить список комментариев в своем состоянии, получая данные от дочернего компонента. Как метод отправки, так и состояние должны войти в новый родительский компонент `CommentBox`. В листинге 2.10 показано, как создать компонент и настроить эти методы.

Листинг 2.10. Обработка отправки комментариев и итерация по элементам

```
...
class CommentBox extends Component {
    constructor(props) {
        super(props);
        this.state = {
            comments: this.props.comments
        };
        this.handleCommentSubmit = this.handleCommentSubmit.bind(this);
    }
    handleCommentSubmit(comment) {
        const comments = this.state.comments;
        // Обратите внимание, что мы не изменяли
        // состояние непосредственно
        comment.id = Date.now();
        const newComments = comments.concat([comment]);
        this.setState({
            comments: newComments
        });
    }
    render() {
        return React.createElement(
            'div',
            {
                className: 'commentBox'
            },
            React.createElement(Post, {
                id: this.props.post.id,
                content: this.props.post.content,
                user: this.props.post.user
            }),
            this.state.comments.map(function(comment) {
                return React.createElement(Comment, {
                    key: comment.id,
                    id: comment.id,
                    content: comment.content,
                    user: comment.user
                });
            }),
            React.createElement(CreateComment, {
```

Передача комментариев на верхнем уровне в CommentBox

Никогда не изменяйте состояние напрямую, вместо этого сделайте копию

Как и прежде, передача переменной данных на самом верхнем уровне, чтобы получить доступ к данным post

Установка соответствия комментариев в this.state.com и возвращение элемента React для каждого из них

```
                onCommentSubmit: this.handleCommentSubmit
            })
        );
    }
}

CommentBox.propTypes = {
    post: PropTypes.object,
    comments: PropTypes.arrayOf(PropTypes.object)
};

const App = React.createElement(CreateComment);

ReactDOM.render(
    React.createElement(CommentBox, {

        comments: data.comments,
        post: data.post
    }),
    node

);
...
```

Передача родительского метода handleCommentSubmit компоненту CreateComment для использования

Передача разметки компоненту CommentBox в качестве свойства

Код из листинга 2.10 доступен по адресу codesandbox.io/s/z6o64oljn4.

На этом этапе у вас есть неприглядный, непроверенный, но функциональный компонент, который выполняет проверку свойств, состояние обновления и позволяет добавлять новые комментарии. Сделано немного, поэтому я дам вам задание — оформить окно комментариев в духе нашей вымышленной компании Letters.

2.4. Знакомство с JSX

Вы создали свой первый динамический компонент React. Если это было несложно, отлично! Если вы обнаружили, что трудно читать фрагменты кода со всеми вложенными элементами `React.createElement`, это тоже хорошо. Мы обсудим некоторые более простые способы создания компонентов, но сначала нужно сосредоточиться на фундаментальных принципах. Изучение остального в обратном порядке (волшебство и легкость вначале, основы и детали позже) обычно намного проще, но в итоге это может вам помешать, потому что вы не поняли, как работает основной механизм. Если взглянете на свою разметку, то вспомните цитату Сэмюэла Джонсона (Samuel Johnson), которая сейчас как нельзя более к месту: *«То, что мы надеемся когда-либо делать с легкостью, сначала должны научиться делать с усердием»*.

2.4.1. Создание компонентов с помощью JSX

Важно овладеть основными принципами этого процесса, но это не значит, что стоит осложнять себе жизнь. Существуют лучшие и более простые средства и способы создания компонентов React, чем использование функции `React.createElement`. Встречайте JSX — одно из таких средств.

Что такое JSX? Это XML-подобное расширение синтаксиса для ECMAScript без какой-либо определенной семантики, предназначенной специально для использования препроцессорами. Другими словами, JSX является расширением JavaScript, похожим на XML, и предназначено оно только для применения инструментами кодирования. В любом случае это не то, что вы увидите в спецификации ECMAScript.

JSX помогает, позволяя вам писать код в стиле XML (похожем на HTML) вместо использования `React.createClass`. Другими словами, он позволяет вам писать код, *похожий* на HTML, но им не являющийся. Препроцессор JSX, такой как компилятор Babel, превращает ваш JavaScript в код, совместимый со старыми браузерами. Он анализирует весь JSX-код и преобразует его в обычный JavaScript-код. Это необходимо, так как выполнение непреобразованного JSX-кода в браузере недопустимо — при анализе JavaScript будут возникать всевозможные синтаксические ошибки.

Перспектива верстки в стиле XML HTML-подобного кода способна вас расстроить, но есть много веских причин работать с JSX, и я чуть позже расскажу про них. А пока ознакомьтесь с листингом 2.11, чтобы узнать, как выглядит компонент комментария в JSX. Я опустил часть кода, чтобы было проще сфокусироваться на синтаксисе JSX. Обратите внимание на то, что компилятор Babel добавлен в среду CodeSandbox. Как правило, вы будете использовать систему сборок типа Webpack, чтобы преобразовать JavaScript-код, также можете импортировать Babel и работать, пропустив сборку. Тем не менее это намного более медленный способ, который нельзя применять на запущенном приложении. Дополнительная информация имеется на сайте babeljs.io.

Код листинга 2.11 доступен по адресу codesandbox.io/s/vnwz6y28x5.

Листинг 2.11. Переработка компонентов с помощью JSX

```
...
    class CreateComment extends Component {
    constructor(props) {
        super(props);
        this.state = {
        content: '',
        user: ''
    };
    this.handleUserChange = this.handleUserChange.bind(this);
    this.handleTextChange = this.handleTextChange.bind(this);
    this.handleSubmit = this.handleSubmit.bind(this);
}
//...
    render() {
        return (
            <form onSubmit={this.handleSubmit} className="createComment">
                <input
                    value={this.state.user}
                    onChange={this.handleUserChange}
                    placeholder="Your name"
                    type="text"
                />
```

Вместо создания свойств на объекте в JSX вы их создаете так же, как и в HTML, — для передачи данных в выражениях используется синтаксис {}

```
                <input
                    value={this.state.content}
                    onChange={this.handleTextChange}
                    placeholder="Thoughts?"
                    type="text"
                />
                <button type="submit">Post</button>
            </form>
        );
    }
}

class CommentBox extends Component {
//...
    render() {
        return (
            <div className="commentBox">
                <Post
                    id={this.props.post.id}
                    content={this.props.post.content}
                    user={this.props.post.user}
                />
                {this.state.comments.map(function(comment) {
                    return (
                        <Comment
                            key={comment.id}
                            content={comment.content}
                            user={comment.user}
                        />
                    );
                })}
                <CreateComment onCommentSubmit={this.handleCommentSubmit}
                />
            </div>
        );
    }
}

CommentBox.propTypes = {
    post: PropTypes.object,
    comments: PropTypes.arrayOf(PropTypes.object)
};

ReactDOM.render(
    <CommentBox
        comments={data.comments}
        post={data.post}
    />,
    node
);

......
```

Это класс React с именем Post, который вы создали ранее, — обратите внимание, что теперь намного понятнее, что это настраиваемый компонент

Используется обычный JavaScript-код внутри скобок {} для итерации по комментариям и создания компонента комментария для каждого

Передача в обработчик handle-CommentSubmit как свойства

На верхнем уровне CommentBox также является настраиваемым компонентом, который вы предоставляете свойствам и передаете в React DOM для рендеринга

2.4.2. Преимущества JSX и отличия от HTML

Теперь, увидев JSX в действии, вы, может быть, отнесетесь к нему менее скептически. Но если все еще не уверены, стоит ли с ним работать, рассмотрите ряд преимуществ, которые он дает в работе с компонентами React.

- *Сходство с HTML и более простой синтаксис.* Если многократное написание функции `React.createElement` кажется утомительным или если вы обнаружили, что за вложенностью трудно уследить, вы не одиноки. Схожесть JSX с HTML упрощает декларирование структуры компонентов и значительно улучшает читаемость.
- *Декларативность и инкапсулированность.* Включив код, который будет составлять ваше представление наряду с любыми связанными методами, вы создаете функциональную группу. По сути, все, что вам нужно знать о компоненте, находится в одном месте. Без необходимости от вас ничего не скрывают, так что вам будет проще разобраться в своих компонентах и понять, как работает эта система.

Возникает ощущение отката к концу 1990-х, когда можно было вставлять свою разметку рядом с JavaScript-кодом, но это не значит, что это плохая идея.

Важно отметить, что JSX не является языком HTML (или XML) — он только компилируется в обычный код React, такой же, какой вы использовали до сих пор, и у него нет точного синтаксиса и соглашений. Существуют тонкие различия, на которые вам нужно обратить внимание. Они будут подробно рассмотрены в следующих главах, а сейчас я кратко упомяну некоторые из них.

- *HTML-теги по сравнению с компонентами React.* Пользовательские компоненты React, созданные вами с помощью функции `React.createClass`, используются по умолчанию, поэтому вы можете определить разницу между пользовательскими и встроенными HTML-компонентами.
- *Атрибутные выражения.* Если вы хотите применить выражение JavaScript в качестве значения атрибута, оберните выражение в пару фигурных скобок (`<Comment a={this.props.b}/>`) вместо кавычек (`<User a="this.props.b"/>`), как показано в листинге 2.8.
- *Логические атрибуты.* Неуказанное значение атрибута (`<Plan active />`, `<Input checked />`) заставляет JSX рассматривать его как истинное. Чтобы передать ложное значение, вы должны использовать выражение (`attribute = {false}`).
- *Вложенные выражения.* Чтобы вложить значение выражения внутрь элемента, применяется также пара фигурных скобок (`<p> {this.props.content} </ p>`).

В JSX есть свои нюансы и даже подводные камни, и в последующих главах будет рассмотрено все это и многое другое. Вы станете широко использовать JSX в своих компонентах и в результате сможете гораздо проще создавать компоненты, управлять ими и взаимодействовать с ними.

2.5. Резюме

Мы потратили много времени на разговор о компонентах в этой главе, поэтому вспомним некоторые ключевые моменты.

- Существует два основных типа элементов для создания компонентов в React, с которыми мы работаем: элементы и классы. Элементы React — это то, что вы хотите видеть на экране, они сопоставимы с DOM-элементами. В то же время классы React являются классами JavaScript, которые наследуются от класса `React.Component`. Это то, что мы обычно называем *компонентами*. Они создаются из обоих классов (обычно расширяя `React.Component`) или функций (функциональные компоненты без состояния, описанные в последующих главах).
- Классы React получают доступ к состоянию, которое может меняться со временем (изменяемое состояние), но все элементы React получают доступ к свойствам, которые изменяться не должны (неизменяемое состояние).
- В классах React также есть специальные методы, называемые *методами жизненного цикла*, которые будут вызваны React в определенном порядке во время процесса рендеринга и обновления. Так ваши компоненты становятся более предсказуемыми в работе и позволяют легко подключить их к процессу обновления.
- В классах React могут использоваться специально определенные для них методы выполнения таких задач, как изменение состояния.
- Компоненты React взаимодействуют через свойства и имеют родственные отношения. Родительские компоненты могут передавать данные потомкам, но потомки не могут изменять предков. Они могут передавать предкам данные через обратные вызовы, но не имеют прямого доступа к ним.
- JSX — это XML-подобное расширение JavaScript, которое позволяет писать компоненты намного проще и более привычным способом. Вначале может показаться странным писать то, что выглядит как HTML в JavaScript, но разметка JSX более привычна и, как правило, легче читаема, чем вызовы функции `React.createElement`.

Вы создали свой первый компонент, но это только верхняя часть айсберга операций, которые выполняются с помощью React. В следующей главе вы начнете работать с более сложными данными, узнаете о различных типах компонентов и по мере расширения знаний о React углубитесь в состояния.

Часть II
Компоненты и данные в React

В первой части мы вкратце рассмотрели React, бегло прошлись по некоторым из его API и создали несколько компонентов. Надеюсь, у вас теперь есть общее представление о том, что такое React и как работает эта технология. Но поверхностная экскурсия не позволит вам научиться использовать все возможности React, чтобы создавать надежные динамические пользовательские интерфейсы.

В части II вы начнете подробно исследовать React и внимательно изучите его API. Мы рассмотрим, как создавать компоненты и некоторые из доступных их типов. В главе 3 изучим, как данные передаются через React-приложение. Это поможет вам понять, как React взаимодействует с данными в компонентах.

В главе 4 будут рассмотрены методы жизненного цикла в React и вы приступите к созданию проекта, с которым не расстанетесь и в дальнейшем, — приложения для социальных сетей под названием Letters Social. Если хотите взглянуть на итоговый проект, посетите сайт social.react.sh. Глава 4 поможет вам понять API React Component и покажет, как настроить систему для создания проекта Letters Social.

В главах 5 и 6 мы рассмотрим формы в React. Формы — важная часть большинства веб-приложений, и мы поговорим о том, как они работают в React. Вы добавите формы в приложение Letters Social и создадите интерфейс, который позволит пользователям писать сообщения и применять сервис Mapbox для добавления меток местоположения.

В главах 7 и 8 мы погрузимся в маршрутизацию. Маршрутизация — еще одна критически важная часть современных приложений веб-интерфейсов. С помощью React вы создадите роутер с нуля и добавите страницы в приложение Letters Social. К концу главы вы интегрируете службу Firebase, чтобы пользователи могли авторизоваться в приложении.

В завершение части II в главе 9 мы сосредоточимся на тестировании. Тестирование — важная часть процесса разработки программного обеспечения, и в этом React не отличается от других платформ. Вы изучите инструменты тестирования Jest и Enzyme для проверки компонентов React.

3 Данные и потоки данных в React

- Изменяемое и неизменяемое состояние.
- Компоненты с сохранением состояния и без него.
- Связь компонентов.
- Однонаправленный поток данных.

Глава 2 была быстрым туром по React. Мы познакомились с React, рассмотрели некоторые концепции, лежащие в основе структуры и API этой библиотеки, и даже разработали простой блок комментариев с компонентами React. В этой главе вы будете более активно работать с компонентами и подготовитесь к созданию проекта Letters Social. Но прежде, чем сделать это, вы должны еще кое-что узнать о том, как работать с данными и потоками данных в React-приложениях.

3.1. Использование состояния

В главе 2 вы лишь краешком глаза увидели, как нужно работать с данными в компонентах React, но если хотите создавать сложные React-приложения, следует уделить этому больше времени. В этом разделе вы узнаете:

- о том, что такое состояние;
- как React обрабатывает состояние;
- как данные передаются через компоненты.

Современные веб-приложения обычно создаются на основе данных. Конечно, существует много статичных сайтов (мой блог ifelse.io — один из них), но даже они обновляются со временем и обычно относятся к другой категории. Большинство веб-приложений, которые люди постоянно используют, очень динамичны и заполнены данными, изменяющимися с течением времени.

Вспомните такое приложение, как Facebook. Так как это социальная сеть, данные являются основой существования этого сайта. Он обеспечивает множество способов взаимодействия с другими людьми через Интернет, и все они реализуются путем изменения и получения данных в вашем браузере или других платформах. Многие

другие приложения содержат невероятно сложные данные, которые должны быть представлены в пользовательском интерфейсе и которые люди смогут понять и легко использовать. Разработчики должны иметь возможность поддерживать и прорабатывать эти интерфейсы и то, как данные передаются через них, поэтому приемы работы приложения с данными так же важны, как и способность обрабатывать изменяющиеся с течением времени данные. Приложение, которое вы начнете разрабатывать в следующей главе, Letters Social (посетите его страницу social.react.sh), будет задействовать множество изменяющихся данных, но создать его не так сложно, как большинство потребительских или бизнес-приложений. В данной главе я расскажу об этом более подробно, и мы продолжим изучать работу с данными в React и в дальнейшем.

3.1.1. Что такое состояние

Сделаем краткий, упрощенный обзор состояния в React, чтобы разобраться в нем. Даже если вы никогда раньше не слышали о состояниях в программе, вы, вероятно, встречались с ними. Большинство написанных вами программ, скорее всего, имели какое-то состояние. Если вы когда-либо работали с интерфейсной инфраструктурой, такой как Vue, Angular или Ember, то почти наверняка создавали пользовательские интерфейсы, у которых был некий показатель состояния. Компоненты React также могут иметь состояние. Но о чем именно мы говорим, используя термин «состояние»? Попробую дать определение.

ОПРЕДЕЛЕНИЕ

> **Состояние** — вся информация, к которой программа имеет доступ в данный момент времени.

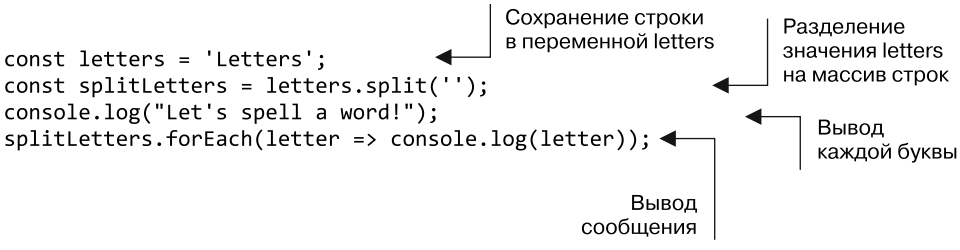
Это упрощенное определение, не учитывающее некоторых тонких нюансов, но хорошо подходящее для наших целей. Учеными было написано множество статей, посвященных точному определению состояния в компьютерных системах, но для нас состояние — это вся информация, к которой имеет доступ программа в данный момент. Она включает в себя, помимо прочего, все значения, на которые вы можете ссылаться, без каких-либо дальнейших присвоений или вычислений в данный момент времени. Другими словами, это мгновенный снимок того, что вы знаете о программе в данное мгновение.

К этой информации относятся, например, любые ранее созданные переменные или другие доступные значения. Когда вы меняете переменную, а не просто извлекаете с ее помощью значение, меняется состояние программы — оно больше не такое, каким было раньше. Вы можете получить состояние в данный момент, только извлекая или получая значения, но когда что-то меняете со временем, состояние программы изменяется. Технически базовое состояние вашего компьютера меняется каждый раз, когда вы его используете, но сейчас мы говорим только о состоянии программы.

Рассмотрим некоторый код и перейдем к упрощенному состоянию программы в следующем листинге. Мы не будем вдаваться во все процессы, которые происходят

за кадром, а просто пытаемся более четко представить в своих программах данные, чтобы проще было понять компоненты React.

Листинг 3.1. Простое состояние программы




```
const letters = 'Letters';
const splitLetters = letters.split('');
console.log("Let's spell a word!");
splitLetters.forEach(letter => console.log(letter));
```

В листинге 3.1 продемонстрирован простой сценарий, который выполняет некоторые базовые задания, манипулирует данными и записывает их в журнал. Он скучен, но мы воспользуемся им, чтобы узнать больше о состояниях. JavaScript применяет так называемую семантику *выполнения до завершения*, что означает: программы будут выполняться потоком сверху вниз в предполагаемом порядке. Движки JavaScript часто оптимизируют код таким образом, которого вы, возможно, не ожидаете, но он все равно должен работать в соответствии с исходным кодом.

Попробуйте прочитать код в листинге 3.1 сверху вниз, по одной строке. Если хотите использовать для этого отладчик браузера, перейдите на сайт codesandbox.io/s/n9mvol5x9p. Должна открыться панель инструментов разработчика для вашего браузера, и вы сможете перейти по каждой строке кода и увидеть все присвоения значений переменным и многое другое.

Для достижения наших целей рассмотрим каждую строку кода как момент времени. Работая с упрощенным определением состояния как всей информации, доступной программе в данный момент времени, как бы вы описали состояние приложения в каждый такой момент? Обратите внимание на то, что мы все упрощаем и пропускаем такие действия, как закрытие, сбор мусора и пр.

- `letters` — это переменная с присвоенной ей строкой `'Letters'`.
- Переменная `splitLetters` создается разделением значения переменной `letters` на буквы, хотя `letters` все еще остается доступной.
- Все данные из шагов 1 и 2 все еще доступны; сообщение отправляется в консоль.
- Программа выполняет итерацию по каждому из элементов массива и выводит символ. Этот процесс, вероятно, будет продолжаться в течение нескольких моментов времени, поэтому в программе есть информация из доступного ей метода `Array.forEach`.

По мере выполнения программы состояние менялось и появилось больше информации, потому что вы ничего не удалили или не изменили ссылки. В табл. 3.1 показано, как доступная информация увеличивается со временем в ходе выполнения программы.

Таблица. 3.1. Состояние шаг за шагом

Шаг	Состояние, доступное программе
1	letters = "Letters"
2	letters = "Letters" splitLetters = ["L", "e", "t", "t", "e", "r", "s"]
3	letters = "Letters" splitLetters = ["L", "e", "t", "t", "e", "r", "s"]
4	letters = "Letters" splitLetters = ["L", "e", "t", "t", "e", "r", "s"] для подшагов от 0 до длины splitLetters: letter = "L" (затем "e", "t" и т. д.)

Попробуйте открыть любой свой код и разобраться, какая информация доступна программе в каждой строке. Мы склонны упрощать код — и это правильно, потому что не нужно думать обо всех функциях сразу, так как даже в простых программах количество информации может быть огромным.

Вывод, на который следует обратить внимание, заключается в том, что, когда работающая программа становится довольно сложной (таковыми могут стать даже самые простые пользовательские интерфейсы), проанализировать ее работу, вероятно, будет трудно. Под этим я подразумеваю, что из-за сложности системы непросто помнить обо всем сразу и логика системы может быть непонятной. Это справедливо для большинства программ, но, когда дело доходит до создания пользовательских интерфейсов, ситуация способна значительно усложниться.

Пользовательский интерфейс современных браузерных приложений часто представляет собой сочетание множества технологий, включая серверные источники данных, API стилизации и разметки, фреймворки JavaScript, API браузера и т. д. Достижения в области пользовательского интерфейса направлены на упрощение этой проблемы, но все же она продолжает существовать. А часто она только увеличивается с ростом ожиданий людей от веб-приложений, так как последние распространяются все шире и внедряются в общество и повседневную жизнь. Чтобы React была полезной, она должна помочь нам, уменьшив чрезвычайную сложность некоторых современных пользовательских интерфейсов или оградив нас от них. Надеюсь, вы поймете, что React действительно делает это. Но как? Одним из способов является предоставление двух специальных API для работы с данными: свойств и состояния.

3.1.2. Изменяемое и неизменяемое состояние

В React-приложениях доступны два основных способа работы с компонентами: через состояние, которое можно изменить, и состояние, которое изменять не следует. Я здесь упрощаю: существует много типов данных и состояний, которые будут использоваться в вашем приложении. Вы можете представлять данные по-разному: как бинарные деревья, карты, наборы или обычные объекты JavaScript. Но способы

взаимодействия с состоянием компонентов React делятся на эти две категории. В React они известны как *состояние* (данные, которые можно изменить внутри компонента) и *свойства* (получаемые компонентом данные, которые он не должен изменять).

Вероятно, вы слышали о состоянии и свойствах, которые называются изменяемыми и неизменяемыми. Это верно лишь отчасти, потому что JavaScript не поддерживает действительно неизменяемые объекты (за исключением, скажем, символов). В компонентах React состояние, как правило, изменяемо, а свойства изменять не следует. Давайте еще немного поговорим об идеях изменяемости и неизменяемости, прежде чем погрузиться в специфические для React API.

В главе 2 вы видели, что при вызове изменяемого состояния мы имеем в виду, что можем перезаписать или обновить эти данные (например, переменную). В то же время неизменяемое состояние нельзя изменять. Существуют также неизменяемые структуры данных, которые могут быть изменены, но только контролируемыми способами (так работает API состояния в React). Когда в главах 10 и 11 начнется работа с Redux, вы будете эмулировать неизменяемые структуры данных.

Мы можем немного расширить понимание изменяемости и неизменяемости, включив в него соответствующие типы структур данных.

- *Неизменяемость* — неизменяемая, постоянная структура данных, с течением времени начинает поддерживать множество версий, но не может быть напрямую перезаписана. Неизменяемые структуры данных, как правило, постоянны.
- *Изменяемость* — изменяемая, эфемерная структура данных, поддерживает только одну версию в конкретный момент времени. Изменяемые структуры данных перезаписываются, когда они меняются и не поддерживают дополнительные версии.

Рисунок 3.1 иллюстрирует эти концепции.

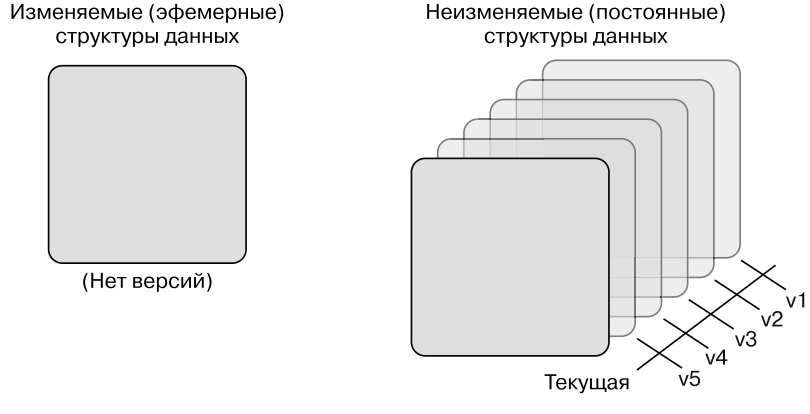


Рис. 3.1. Постоянство и эфемерность в неизменяемых и изменяемых структурах данных. Неизменяемые, или постоянные, структуры данных обычно записывают историю и не меняются, а скорее создают версии того, что изменилось с течением времени. Эфемерные же структуры данных обычно не записывают историю и очищаются при каждом обновлении

Еще один способ понять разницу между неизменяемыми и изменяемыми структурами данных — подумать о каждой из них, как об имеющей разные возможности или воспоминания. Эфемерные структуры данных способны хранить ценные в конкретный момент данные, тогда как постоянные структуры данных могут отслеживать изменения, возникающие с течением времени. Именно здесь становится яснее постоянство неизменяемых структур данных: сделаны только копии состояния — они не заменяются. Старое состояние заменяется новым, но данные неизменны. На рис. 3.2 показано, как происходят изменения.

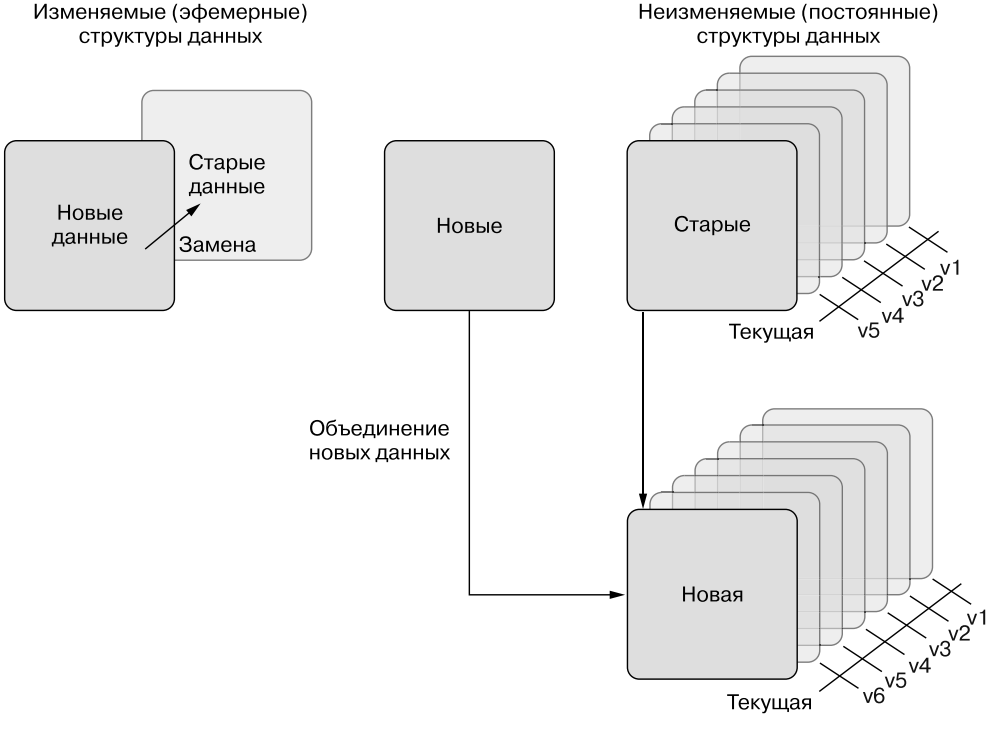


Рис. 3.2. Обработка изменений при наличии изменяемых и неизменяемых данных. Эфемерные структуры данных не имеют версий, поэтому, когда вы вносите в них изменения, все предыдущее состояние сбрасывается. Можно сказать, что они существуют в данный момент, тогда как неизменяемые структуры данных могут сохраняться с течением времени

СОВЕТ

Другой подход к неизменяемости в сравнении с изменяемостью — посмотреть на различия между командами Сохранить и Сохранить как. Во многих компьютерных программах вы можете сохранить файл, заменив исходный с тем же именем, или сохранить его копию под другим именем. Неизменяемость аналогична, так как при сохранении вы сохраняете копию, тогда как изменяемые данные могут быть перезаписаны.

Несмотря на то что изначально JavaScript не поддерживает по-настоящему неизменяемые структуры данных, React позволяет изменить состояние компонента (с помощью интерфейса `setState`) и свойства, доступные только для чтения. Вообще-то неизменяемость и неизменяемые структуры данных в целом имеют гораздо больше возможностей, но нам они не понадобятся. Если все же интересно узнать о них, то скажу: проводился целый ряд научных исследований этого вопроса. Существуют также способы широко использовать неизменяемые структуры данных в ваших приложениях JavaScript (React и не только) с помощью таких библиотек, как Immutable JS (для получения дополнительной информации см. страницу facebook.github.io/immutable-js/), но в React мы имеем дело только с API состояния и свойств.

3.2. Состояние в React

Вы узнали немного больше о состоянии, неизменяемости и изменяемости. Как все это вписывается в React? В предыдущей главе вы уже встречались с API свойств и состояния, поэтому, вероятно, догадываетесь, что это важные части создаваемых компонентов. Фактически это два основных способа взаимодействия компонентов React с данными и общения друг с другом.

3.2.1. Изменяемое состояние в React: состояние компонента

Начнем с API состояния. Хотя можно сказать, что у всех компонентов есть некое состояние (общая концепция), не все компоненты в React имеют локальное состояние. В дальнейшем, когда я говорю о состоянии, я имею в виду React API, а не общую концепцию. Компоненты, которые наследуются от класса `React.Component`, получат доступ к этому API. React будет создавать и отслеживать экземпляр поддержки для компонентов, созданных таким образом. Они также получат доступ к ряду методов жизненного цикла, описанных в следующей главе.

Доступ к состоянию компонентов, которые наследуют `React.Component`, вы получаете с помощью метода `this.state`. В этом случае `this` относится к экземпляру класса, а `state` — это особое свойство, которое React будет отслеживать автоматически. Вы можете подумать, что реально обновить `state`, просто назначив ему свойство или изменив его, но это не так. Рассмотрим пример изменения состояния простого компонента React в следующем листинге. Можно создать этот код на локальном компьютере или, что проще, выполнить на странице codesandbox.io/s/ovxpmn340y.

В листинге 3.2 создается простой компонент, который при нажатии на кнопку будет раскрывать секретное имя с помощью интерфейса `setState` для обновления состояния компонента. Обратите внимание на то, что интерфейс `setState` доступен для `this`, потому что компонент наследуется от класса `React.Component`.

Когда вы нажмете на кнопку мыши, будет запущено событие щелчка и выполнена запрограммированная вами функция. После этого будет вызван метод `setState` с объектом в качестве параметра. У этого объекта есть свойство `name`, которое указывает на строку. React запланирует обновление состояния. Когда эта работа будет выполнена, `React DOM` при необходимости обновит DOM. Функция `render` будет вызываться снова, но на этот раз с другим значением, доступным синтаксису выражения JSX ({}), который содержит `this.state.name`. Она прочитает «Mark» вместо «top secret!», и мои секретные идентификационные данные будут отображены!

Листинг 3.2. Использование setState для изменения состояния компонента

```
import React from "react";
import { render } from "react-dom";
class Secret extends React.Component{
  constructor(props) {
    super(props);
    this.state = {
      name: 'top secret!',
    };
    this.onButtonClick = this.onButtonClick.bind(this);
  }
  onButtonClick() {
    this.setState(() => ({
      name: 'Mark'
    }));
  }
  render() {
    return (
      <div>
        <h1>My name is {this.state.name}</h1>
          <button onClick={this.onButtonClick}>reveal the secret!</button>
      </div>
    )
  }
}

render(
  <Secret/>,
  document.getElementById('root')
);
```

Установка начального состояния компонента, чтобы попытки получить к нему доступ с помощью функции render() не возвращали неопределенные значения или не выбрасывали ошибки

Создание компонента React, который будет иметь доступ к постоянному состоянию компонента все время — не забудьте привязать методы класса к экземпляру компонента

Наш первый взгляд на setState, специальный API для изменения состояния компонента; вызовем setState с функцией обратного вызова, которая возвращает новый объект состояния

Привязка функции определения имени к событию щелчка кнопкой мыши

Визуализация компонентов верхнего уровня в HTML-элемент на самом верхнем уровне приложения — укажите свой контейнер на выбор; React DOM сможет его найти

Обычно интерфейс `setState` используется экономно, если есть такая возможность, чтобы повысить производительность и уменьшить сложность (React должна отслеживать и другие данные). Существуют шаблоны, очень популярные в сообществе React, которые позволяют программистам редко задействовать состояние компонента (в том числе Redux, Mobx, Flux и др.), и они прекрасно подойдут как альтернатива для вашего приложения (рассмотрим Redux в главах 10 и 11). Хотя рекомендуется использовать функциональный компонент без состояния или такой шаблон, как Redux, применение интерфейса `setState` само по себе неплохо. В React это по-прежнему основной API для изменения данных в компоненте.

Прежде чем двигаться дальше, важно отметить, что вы никогда не должны напрямую модифицировать `this.state` в компонентах React. Если попытаетесь изменить `this.state` напрямую, вызов функции `setState()` впоследствии может заменить внесенное изменение и, что еще хуже, React не будет иметь никакого представления о сделанном изменении. Даже если вы предполагаете, что способны изменить состояние компонента, то должны рассматривать объекты `this.state` так, как если бы они были неизменяемыми в ваших компонентах (например, свойствах).

Это важно еще и потому, что функция `setState()` не сразу изменяет `this.state`. Вместо этого она создает *переходное* состояние (подробнее о рендеринге и обнаружении изменений — в главе 4). Доступ к `this.state` после вызова этого метода может потенциально вернуть существующее значение. Все это потенциально усложняет отладку, поэтому используйте функцию `setState()` для изменения состояния компонента.

В небольшом взаимодействии, продемонстрированном в листинге 3.2, происходит совсем немногое. Мы продолжим разбирать шаги, которые выполняет React для обновления компонентов, в следующих главах, а пока важно внимательнее изучить метод `render`. Обратите внимание на то, что, даже если вы изменили состояние и данные, это было довольно понятно и предсказуемо.

Или, как вариант, вы можете захотеть, чтобы внешний вид и структура компонента всегда выглядели одинаково. Вам не нужно выполнять уйму лишней работы для двух разных состояний, в которых он мог бы существовать (с открытым секретным именем или без него). React обрабатывал бы все процедуры привязки и обновления основного состояния, а вам требовалось бы только сообщить: «Имя должно быть здесь». React позволяет не задумываться о состоянии в каждый момент времени, как было в подразделе 3.1.1.

Взглянем на интерфейс `setState` чуть пристальнее. Это основной способ изменения динамического состояния в компонентах React, и вы будете часто использовать его в своем приложении. Рассмотрим сигнатуру метода, чтобы понять, что ему нужно передать:

```
setState(
    updater,
    [callback]
) -> void
```

Интерфейс `setState` использует функцию, предназначенную для установки нового состояния компонента и дополнительной функции `callback`. Функция `updater` имеет следующий синтаксис:

```
(prevState, props) => stateChange
```

В предыдущих версиях React вы могли передать интерфейсу `setState` объект вместо функции в качестве первого аргумента. Ключевое отличие от текущих версий React (16 и выше) заключается в том, что интерфейс `setState` был синхронным по своей природе, тогда как React планирует изменение состояния. Формат `callback` лучше передает эту идею и, как правило, лучше согласуется с общими парадигмами декларативности и асинхронности React: вы разрешаете системе (React) планировать обновления, гарантируя порядок, но не время. Это соответствует более декларативному подходу к пользовательскому интерфейсу, и, как правило, гораздо проще думать об этом, чем требовать императивного определения обновлений данных в разное время (часто источник условий гонки).

Если необходимо выполнить обновление до состояния, которое зависит от текущего состояния или свойств, можете получить доступ к ним через аргументы `prevState` и `props`. Зачастую это полезно, когда вы хотите сделать что-то вроде логического переключения и вам нужно знать точное последнее значение, существовавшее перед выполнением обновления.

Сосредоточимся на механике `setState`. Используя объект, возвращенный из функции `updater`, `setState` выполняет мелкое слияние в текущее состояние. Это означает, что вы можете передать объект, а React совместит свойства верхнего уровня объекта с состоянием. Например, допустим, что у вас есть объект со свойствами `A` и `B`. У `B` имеются глубоко вложенные свойства, а `A` — просто строка (`'hi!'`). Поскольку выполняется мелкое слияние, будут сохраняться только свойства верхнего уровня и то, на что они ссылаются, а не все составляющие `B`. React не обнаружит глубоко вложенного свойства `B`, чтобы его обновить. Один из способов обойти это ограничение — сделать копию объекта, глубоко обновить его, а затем применять. У вас есть возможность работать с библиотекой типа `immutable.js` (facebook.github.io/immutable-js/), чтобы упростить работу со структурами данных в React.

`setState` — простой в использовании API: вы передаете компоненту ReactClass некоторые данные для соединения с текущим состоянием, и React обработает его автоматически. Если вам необходимо отследить процесс завершения, подключитесь к нему с помощью дополнительной функции `callback`. В листинге 3.3 показан пример слияния `setState` в действии. Как и прежде, легко создать и запустить свой компонент React на сайте CodeSandbox по адресу codesandbox.io/s/0myo6ny4ww. Это избавит вас от необходимости устанавливать программное обеспечение на своем компьютере.

Пренебрежение мелкими слияниями часто становится источником ошибок при изучении React. В этом примере при нажатии кнопки свойство `name`, вложенное в ключ `user` исходного состояния, будет перезаписано, поскольку оно не существует в новом состоянии. Вы хотели сохранить оба состояния, но одно из них переписало другое.

Листинг 3.3. Мелкое слияние с setState

```
import React from "react";
import { render } from "react-dom";
class ShallowMerge extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      user: {
        name: 'Mark', //
        colors: {
          favorite: '',
        }
      }
    };
    this.onButtonClick = this.onButtonClick.bind(this);
  }
  onButtonClick() {
    this.setState({
      user: { //
        colors: {
          favorite: 'blue'
        }
      }
    });
  }
  render() {
    return (
      <div>
        <h1>My favorite color is {this.state.user.colors.favorite} and my
        name is {this.state.user.name}</h1>
        <button onClick={this.onButtonClick}>show the color!</button>
      </div>
    )
  }
}

render(
  <ShallowMerge />,
  document.getElementById('root')
);
```

В исходном состоянии имя существует в свойстве пользователя...

...но не в состоянии, которое вы настраиваете, — если бы уровень был выше, то мелкое слияние не сработало бы

Упражнение 3.1. Размышления об API setState

В этой главе рассказывается об API React для управления состоянием в компонентах. Ранее упоминалось, что нужно изменять состояние через API setState, а не напрямую. Как вы думаете, почему это станет проблемой и не будет работать? Попробуйте запустить код на странице codesandbox.io/s/j7p824jxnw.

3.2.2. Неизменяемое состояние в React: свойства

Мы говорили о том, как React позволяет работать с данными изменяемым способом через состояние и интерфейс `setState`. А как насчет неизменяемых данных? В React свойства являются основным способом передачи неизменяемых данных. Любой компонент может получить свойства (не только наследуемые от `React.Component`) и использовать их в своих методах `constructor`, `render` и `lifecycle`.

Свойства в React более или менее неизменяемы. Можете задействовать библиотеки и другие инструменты для эмуляции неизменяемых структур данных в своих компонентах, но API свойств React сам по себе является полунеизменяемым. React использует встроенный JavaScript-метод `Object.freeze` (developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Object/freeze), если он доступен, чтобы предотвратить добавление новых свойств или удаление существующих. Кроме того, `Object.freeze` предотвращает изменение существующих свойств (или их перечисляемость, конфигурируемость, возможность записи) и прототипа. Это долгий путь к тому, чтобы вы не изменяли объект `props`, но последний не является технически реально неизменяемым объектом (хотя и кажется таковым).

Свойства — это данные, которые передаются компонентам React либо из родителя, либо из статического метода `defaultProps` самого компонента. При этом состояние компонента локализовано для одного компонента, свойства обычно передаются из родительского компонента. Если вы думаете: «Могу ли я использовать состояние родительского компонента для передачи свойств дочернему?» — вы на правильном пути. Состояние одного компонента может быть свойством другого.

Свойства обычно передаются в JSX как атрибуты, но если вы применяете метод `React.createElement`, то передайте их непосредственно в дочерний компонент через этот интерфейс. Можно передать другому компоненту как свойство любые допустимые данные JavaScript и даже другие компоненты (которые, в конце концов, являются классами). После того как свойства переданы в компонент для использования, вы не должны изменять их изнутри него. Конечно, если попробуете, то, вероятно, получите приятную ошибку типа `Uncaught TypeError: Cannot assign to read-only property '<myProperty>' of object '#<Object>'`. Или, что еще хуже, ваше React-приложение не будет работать должным образом из-за несоответствия ожиданиям.

В листинге 3.4 в следующем подразделе показаны некоторые способы доступа к свойствам и то, как их не следует назначать. Как отмечалось ранее, свойства могут меняться со временем, но не изнутри компонента. Это часть одностороннего потока данных — эта тема затронута в последующих главах. Короче говоря, *односторонний* означает, что измененные данные перемещаются потоком по компонентам от родителей к потомкам. Родительский компонент, использующий состояние (наследующий от `React.Component`), способен изменять его, и измененное состояние может быть передано как свойства дочерним, таким образом изменяя свойства.

Упражнение 3.2. Вызов setState в методе render

Мы установили, что `setState` — это способ обновления состояния компонента. Где вы можете вызвать `setState`? В следующей главе рассмотрим, в каких точках жизненного цикла компонента вызывается `setState`, но теперь сосредоточимся только на методе `render`. Как вы думаете, что произойдет, если `setState` вызвать в методе `render` компонента? Попробуйте на странице codesandbox.io/s/48zv2nwqww.

3.2.3. Работа со свойствами: PropTypes и свойства по умолчанию

Для работы со свойствами существует несколько API, способных помочь во время разработки: PropTypes и свойства по умолчанию. PropTypes обеспечивает функциональность проверки типов, которой вы можете указать, какие свойства компонент будет ожидать в ходе применения. Можете задать типы данных и даже сообщить пользователю компонента, какую форму данных они должны предоставить (например, объект с пользовательским свойством, имеющим определенные ключи). В предыдущих версиях React PropTypes был частью основной библиотеки React, но теперь он распространяется независимо в качестве пакета `prop-types` (github.com/facebook/prop-types).

Библиотека `prop-types` лишена магии — это набор функций и свойств, которые могут помочь выполнить проверку типов на входе. И она неуникальна для React — вы можете использовать ее в другой библиотеке, где хотели бы выполнить проверку типов на входе. Например, включите `prop-types` в другую компонентную платформу, подобную React, например Preact (preactjs.com), и работайте с ней аналогично.

Чтобы установить PropTypes для компонента, вы предоставляете статическое свойство в классе `propTypes`. Обратите внимание на то, что в листинге 3.4 имя статического свойства, которое вы задали для класса компонента, написано в нижнем регистре, тогда как имя объекта, к которому обращаетесь из библиотеки `prop-types`, имеет верхний регистр (`PropTypes`). Чтобы указать необходимые компоненту свойства, вы добавляете имя свойства, которое хотите проверить, и присваиваете ему свойство из экспорта по умолчанию библиотеки `prop-types` (`import PropTypes from 'prop-types'`). Используя PropTypes, можете объявить о любых типе, форме и типе требований (опциональных или обязательных) для свойств.

Еще один инструмент, подходящий для упрощения процесса разработки, — это свойства по умолчанию. Помните, что можно задать исходное состояние компоненту с помощью конструктора классов? Что-то подобное делается и для свойства. Можно предоставить статическое свойство, называемое `defaultProps`, чтобы передать компоненту свойства по умолчанию.

Свойства по умолчанию могут обеспечить ваш компонент всем необходимым, даже если тот, кто применяет его, забывает предоставить ему свойство. В следующем

листинге показан пример использования PropTypes и свойств по умолчанию в компоненте. Запустите код на странице codesandbox.io/s/31ml5pmk4m.

Листинг 3.4. Неизменяемые свойства компонентов React

```
import React from "react";
import { render } from "react-dom";
import PropTypes from "prop-types";

class Counter extends React.Component {
  static propTypes = {
    incrementBy: PropTypes.number,
    onIncrement: PropTypes.func.isRequired
  };
  static defaultProps = {
    incrementBy: 1
  };

  constructor(props) {
    super(props);
    this.state = {
      count: 0
    };
    this.onButtonClick = this.onButtonClick.bind(this);
  }
  onButtonClick() {
    this.setState(function(prevState, props) {
      return { count: prevState.count + props.incrementBy };
    });
  }
  render() {
    return (
      <div>
        <h1>{this.state.count}</h1>
        <button onClick={this.onButtonClick}>++</button>
      </div>
    );
  }
}

render(<Counter incrementBy={1} />, document.getElementById("root"));
```

Указание объекта с «формой»

Вы можете связать любые propTypes с isRequired, чтобы убедиться, что выводится предупреждение, если свойство не отображается

3.2.4. Функциональные компоненты без состояния

Как поступить, если нужно создать простой компонент, который использует только свойства, но не состояние? Оказывается, это распространенный случай, особенно с некоторыми поддерживаемыми приложениями React архитектурными программными шаблонами, которые мы рассмотрим позже, например Flux и Redux. В этих случаях часто требуется сохранять состояние централизованно, а не распылять-ся по компонентам. Применение только свойств полезно и в других ситуациях. Было бы неплохо задействовать меньше ресурсов для вашего приложения, если React не должна управлять экземпляром поддержки самостоятельно.

Как оказалось, существует тип компонента, который реально создать и который использует только свойства, — функциональный компонент без состояния. Разработчики иногда называют их компонентами *без состояния*, *функциональными* компонентами и другими подобными именами, так что будет непонятно, о чем идет речь. Они обычно означают одно и то же: компонент React, который не наследуется от `React.Component` и, следовательно, не получает доступа к состоянию компонента или другим методам жизненного цикла.

Неудивительно, что функциональный компонент без состояния является просто компонентом, который не имеет доступа к API React (или к другим методам, унаследованным от `React.Component`) и не способен его применять. Он не имеет состояния не потому, что у него нет состояния вообще, а потому, что не получает экземпляра поддержки, которым React будет управлять самостоятельно. Это означает, что у него отсутствуют методы жизненного цикла (см. главу 4), нет состояния компонента и потенциально меньшие затраты памяти.

Компоненты без состояния функциональны, поскольку могут быть записаны как именованные функции или выражения анонимных функций, назначенные переменной. Они принимают только свойства и, поскольку возвращают один и тот же результат на основе заданного ввода, в основном считаются чистыми. Это обусловливает их скорость, так как React потенциально сможет выполнить оптимизацию, избегая ненужных проверок жизненного цикла или распределения памяти. В листинге 3.5 показан простой пример функционального компонента без состояния. Запустите код на странице codesandbox.io/s/l756002969.

Листинг 3.5. Функциональные компоненты без состояния

```
import React from "react";
import { render } from "react-dom";
import PropTypes from "prop-types";

function Greeting(props) {
  return <div>Hello {props.for}!</div>;
}

Greeting.propTypes = {
    for: PropTypes.string.isRequired
};

Greeting.defaultProps = {
    for: 'friend'
};

render(<Greeting for="Mark" />, mountNode);

// Или используйте функцию ниже
// const Greeting = (props) => <div>Hello {props.for}</div>;
// ...указав свойства или свойства по умолчанию так же, как и раньше
// render(<Greeting name="Mark" />, document.getElementById("root"));
```

Для любой формы функционального компонента без состояния вы можете указать propTypes и свойства по умолчанию в качестве свойств для функции или переменной

Функциональные компоненты без состояния могут быть созданы с функциями или анонимными функциями

Функциональные компоненты без состояния могут быть мощными, особенно при применении в сочетании с родительским компонентом, имеющим экземпляр поддержки. Вместо того чтобы устанавливать состояние для нескольких компонентов, можете создать единый родительский компонент, имеющий состояние, и использовать простые дочерние компоненты для всего остального. В главах 10 и 11 рассмотрим Redux для перехода этого шаблона на совершенно новый уровень. В React-приложениях, работающих с Redux, вы обычно создаете меньше компонентов с состоянием (хотя иногда это все еще имеет смысл), взамен централизуя состояние в одном расположении (хранилище).

Упражнение 3.3. Использование состояния одного компонента для изменения свойств другого

В этой главе рассказывалось о свойствах и состоянии как основных способах работы и обмена данными в компонентах React. Вы никогда не должны напрямую изменять состояние или свойство, но с помощью `setState` можете потребовать от React обновить состояние компонента. Как бы вы использовали состояние одного компонента для изменения свойств другого? Посетите страницу codesandbox.io/s/38zq71q75, чтобы попрактиковаться.

3.3. Связь компонентов

Создав простой компонент комментариев, вы увидели, что можете получать одни компоненты из других. Это одно из преимуществ React. Легко также создавать другие компоненты из подкомпонентов, сохраняя при этом код в полном порядке. Легко выразить отношения *«есть»* и *«содержит»* между компонентами. Это означает — представить компоненты как *содержащие* некоторую часть, *будучи* определенной сущностью.

Замечательно, что вы можете смешивать и комбинировать компоненты и гибко строить что-либо, но как заставить их взаимодействовать друг с другом? Многие фреймворки и библиотеки предлагают специфический метод, позволяющий разным частям приложения работать сообща. Вероятно, вы слышали о существующей в Angular.js или Ember.js услуге связи между различными частями приложения или получали ее. Обычно это широкодоступные долгоживущие объекты, где можно хранить состояние и получать доступ к нему из разных частей приложения.

Использует ли React службы или что-то подобное? Нет. Если нужно, чтобы в React компоненты связывались друг с другом, вы передаете свойства, выполняя два простых действия:

- разрешаете доступ к данным в родителе (состояние или свойство);
- передаете эти данные дочернему компоненту.

В листинге 3.6 показаны примеры отношений «родитель — потомок», с которыми вы знакомы, и отношений «владелец — собственник». Запустите код на странице codesandbox.io/s/pm18mlz8jm.

Листинг 3.6. Передача свойств от родителя к потомку

```
import React from "react";
import { render } from "react-dom";
import PropTypes from "prop-types";

const UserProfile = props => {
  return <img src={'https://source.unsplash.com/user/${props.username}'} />;
};
UserProfile.propTypes = {
  pagename: PropTypes.string
};

UserProfile.defaultProps = {
  pagename: "erondu"
};

const UserProfileLink = props => {
  return <a href={'https://ifelse.io/${props.username}'}>{
    props.username}</a>;
};

const UserCard = props => {
  return (
    <div>
      <UserProfile username={props.username} />
      <UserProfileLink username={props.username} />
    </div>
  );
};

render(<UserCard username="erondu" />, document.getElementById("root"));
```

Создание функционального компонента без состояния, который возвращает указанное изображение

Помните, что вы по-прежнему можете указывать свойства по умолчанию и propTypes даже для функциональных компонентов без состояния

UserCard является родителем для UserProfile и UserProfileLink

3.4. Однонаправленный поток данных

Если вы раньше разрабатывали веб-приложения с использованием фреймворков, то, вероятно, слышали о *двунаправленной привязке данных*. Связывание данных — это процесс, который устанавливает соединение между пользовательским интерфейсом приложения и другими данными. На практике это зачастую похоже на библиотеку или структуру, связывающие данные приложения, такие как модели (пользователь) с пользовательским интерфейсом, и синхронизирующие их. Они синхронизированы и таким образом связаны друг с другом. Еще один способ подумать об этом, который будет более полезен в React, — это *проекция*: UI — это данные, проецируемые в представление, и когда они изменяются, изменяется вид (рис. 3.3).

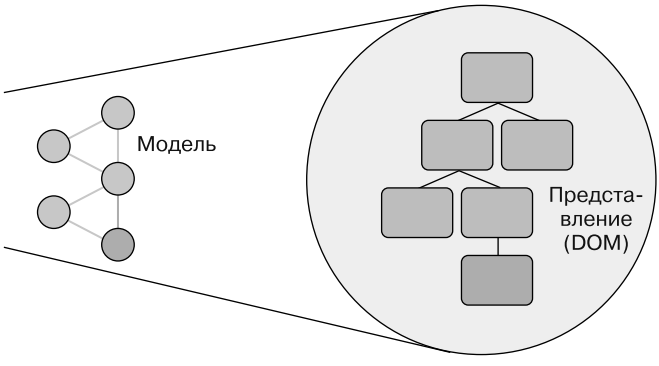


Рис. 3.3. Связывание данных обычно относится к процессу настройки соединения между данными в приложении и представлением (отображением этих данных). Можно представить этот процесс как проекцию данных на что-то, что пользователь видит (например, представление)

Еще одна ассоциация с привязкой данных — *поток* данных: как они перемещаются через разные части вашего приложения? По сути, вы спрашиваете: «Что может обновить что-то, откуда и как?» Важно понять, как применяемые инструменты формируют данные, управляют ими и перемещают их, если вы хотите их использовать эффективно. Различные библиотеки и фреймворки задействуют разные подходы к потоку данных (React также имеет собственный подход).

В React данные передаются в одном направлении. Это означает, что вместо горизонтального потока между объектами, где каждый способен обновлять каждого, устанавливается иерархия. Вы можете передавать данные через компоненты, но не должны выходить за рамки и изменять состояние или свойства других компонентов без передачи свойств. А еще у вас не получится изменить данные в родительском элементе.

Однако возможно передавать данные вверх по иерархии через обратные вызовы. Когда родитель получает обратный вызов от дочернего компонента, он способен изменять свои данные и отправлять измененные данные обратно дочерним компонентам. Даже в случае с обратными вызовами данные по-прежнему в целом передаются вниз и все так же определяются родителем, который делает это. Вот почему мы говорим, что в React данные передаются однонаправленно (рис. 3.4).

Однонаправленный поток особенно полезен при создании пользовательских интерфейсов, поскольку упрощает представление о том, как данные перемещаются по приложению. Благодаря иерархии компонентов и тому, как свойства и состояние локализованы для них, обычно проще предсказать, как данные перемещаются по приложению.

В некоторых случаях может показаться, что лучше избегать этой иерархии и иметь право изменять все, что хотите, из любой части приложения, но на практике из-за этого приложения будет трудно поддерживать, а при отладке могут возникнуть сложные ситуации. В последующих главах будут рассмотрены архитектурные мо-

дели, такие как Flux и Redux, которые позволят поддерживать однонаправленную парадигму потока данных, координируя действия, которые могут возникать в компонентах или приложении.

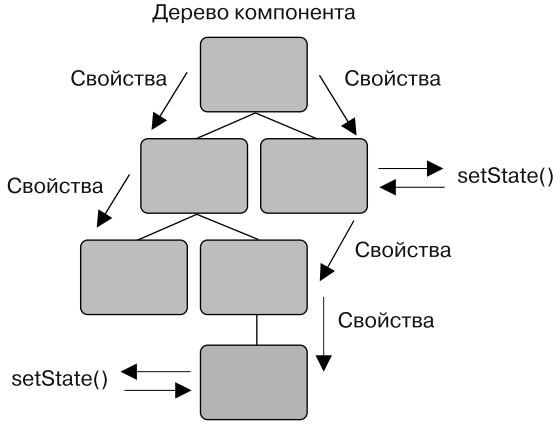


Рис. 3.4. Данные передаются в одном направлении. Свойства передаются от родителя к потомку (от владельца к собственнику), и потомки не могут редактировать состояние или свойства родительского компонента. Каждый компонент, который имеет экземпляр поддержки, способен изменять свое состояние, но не может изменять ничего вне себя, кроме настройки свойств одного из своих дочерних элементов

3.5. Резюме

В этой главе обсуждались следующие темы.

- Состояние — это вся информация, доступная программе в данный момент.
- Постоянное состояние не изменяется, а изменяемое — изменяется.
- Постоянные, неизменяемые структуры данных не меняются — они только записывают изменения и копируют сами себя.
- Эфемерные, изменяемые структуры данных обнуляются при обновлении.
- React использует как изменяемые (локальное состояние компонента), так и псевдонеизменяемые данные (свойства).
- Свойства являются псевдонеизменяемыми и не должны изменяться после установки.
- Состояние компонента отслеживается экземпляром поддержки и может быть изменено с помощью интерфейса `setState`.
- Интерфейс `setState` выполняет мелкое слияние данных и обновляет состояние компонента, сохраняя любые свойства верхнего уровня, которые не перезаписываются.

- Данные передаются в обратном направлении, от родителей потомкам. Потомки могут возвращать данные родителям через обратный вызов, но не могут напрямую изменять состояние родителя, и родитель не может напрямую изменять состояние потомка. Вместо этого взаимодействие компонентов выполняется через свойства.

В главе 4 я продолжу пополнять ваши знания о состоянии в React и расскажу, как использовать методы жизненного цикла, чтобы подключиться к процессу рендеринга и обновления React. Мы также начнем исследовать обнаружение изменений в React, и вы подготовитесь к созданию приложения Letters Social, применяя свежеполученные навыки!

4 Рендеринг и методы жизненного цикла в React

- Настройка с помощью репозитория приложений.
- Процесс рендеринга.
- Методы жизненного цикла.
- Обновление компонентов React.
- Создание ленты новостей с помощью React.

В этой главе вы сведете вместе изученные концепции и полученные навыки, чтобы создать первое React-приложение. В предыдущих главах мы говорили о работе с данными в React и о разных способах работы с изменяемыми (модифицируемыми) и неизменяемыми (немодифицируемыми) данными. Но для создания еще более надежных компонентов нужно использовать полный API компонента, углубиться в методы жизненного цикла и узнать о процессе рендеринга в React.

Мы рассмотрим *рендеринг* — процесс, посредством которого React превращает ваши данные в пользовательский интерфейс, и некоторые способы взаимодействия с компонентом в течение его жизненного цикла, называемые *методами жизненного цикла*. Вы объедините эту информацию с полученными знаниями о чтении и изменении данных в React (свойства и состояние), обновлении состояния вашего компонента и передаче данных различным компонентам.

4.1. Начало работы с репозиторием Letters Social

В этой главе вы подготовитесь к созданию приложения Letters Social. Притворимся, что вы стартап, сосредоточившийся на разработке грандиозного приложения для социальных сетей. Ваша компания Letters, изобретательно названная так, чтобы отличаться от веб-гигантов типа Alphabet, работает с социальными сетями. Вы станете применять библиотеку React для разработки этого приложения на протяжении всей книги. К концу приложение Letters Social будет рендериться на стороне сервера. Приложение, показанное на рис. 4.1, поддерживает несколько функций, которые вы будете разрабатывать на протяжении книги:

- создание сообщений с текстом;
- добавление в сообщения места расположения с помощью Mapbox;

- добавление к сообщениям лайков и комментариев;
- обеспечение аутентификации OAuth через GitHub и Firebase;
- отображение сообщений в ленте новостей;
- разбивка на страницы (пагинация).

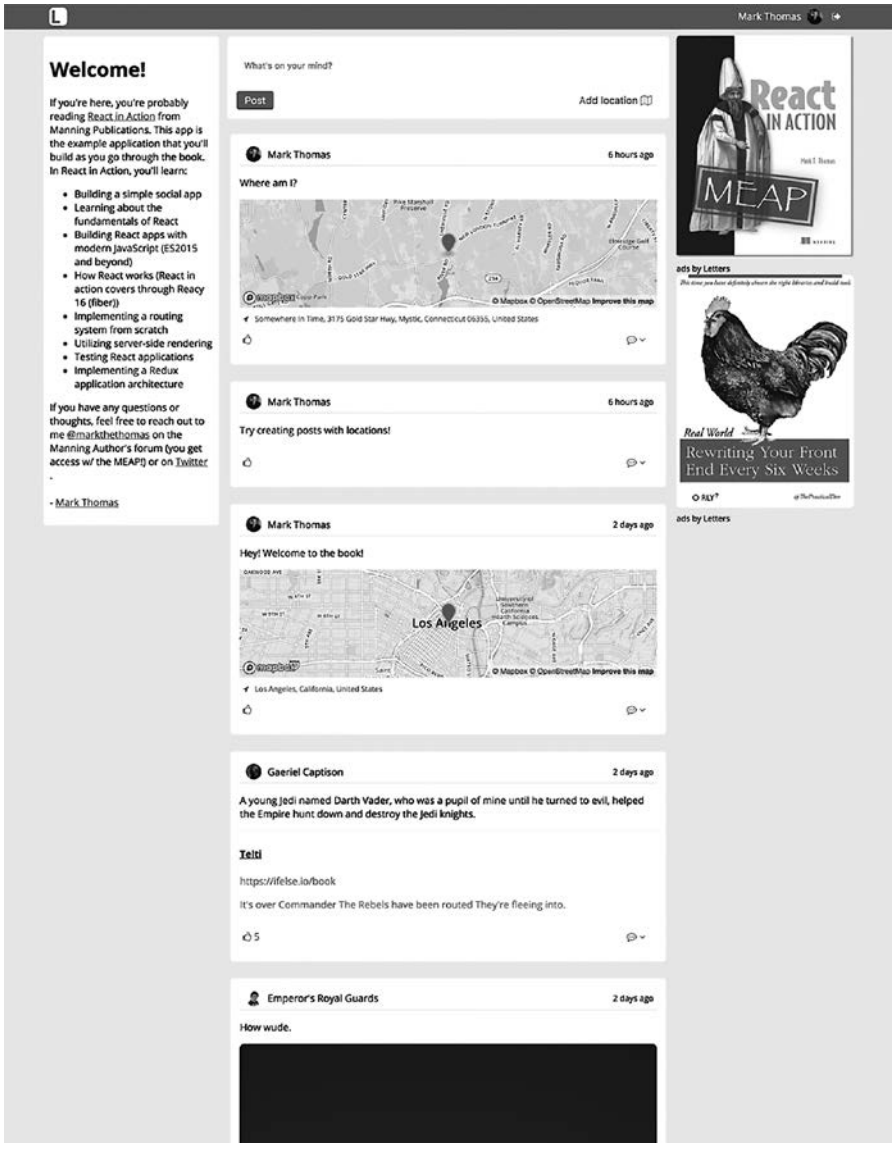


Рис. 4.1. Letters Social — React-приложение, которое вы создадите в этой книге. Ознакомьтесь с его исходным кодом на странице github.com/react-in-action/letters-social и проверьте работу на странице social.react.sh

Мы рассмотрим каждую из этих функций в данной и последующих главах. Чтобы упростить работу, я создал ветвь Git для глав 4–12. Для каждой главы (в некоторых случаях — пары глав) представлен код, который приведен в конце соответствующей главы. Например, если вы откроете ветвь Git для глав 5 и 6, у вас будет код, опубликованный в конце этих глав. Так вы сможете заранее просмотреть проект, начав с любой главы. Если вы хотите работать с главой 9 (например, для изучения приемов тестирования React-приложений), ознакомьтесь с кодом из глав 7 и 8 и начните с нужного момента. Я попытался упростить ознакомление с кодом, а вы задействуйте репозиторий и ветви Git так, как вам хочется. Создавайте запросы на получение данных с вопросами или используйте их в качестве исходного места расположения для новых свойств, которые хотите добавить приложению.

Вы также можете прочитать документацию об исходном коде на странице docs.react.sh. Она не охватывает всех нюансов, но если вам требуется получить представление о коде в стиле JSDoc, это будет отличный старт. В файле README репозитория также перечислен ряд полезных ресурсов. И не стесняйтесь обращаться ко мне, если у вас возникают вопросы (или вам просто нравится книга!). Можете сделать это так, как указано в файле `README`.

4.1.1. Получение исходного кода

Чтобы получить исходный код, перейдите на страницу github.com/react-in-action/letters-social. Это репозиторий, в котором хранится весь исходный код, связанный с книгой. В разделе React in Action сайта GitHub есть несколько других репозиториев, можете проверять и их. Основной исходный код находится по адресу github.com/react-in-action/letters-social. Зайдите туда и либо загрузите исходный код, либо используйте следующие команды для клонирования репозитория:

```
git clone git@github.com:react-in-action/letters-social.git
```

```
git checkout chapter-4
```

Таким образом вы клонируете репозиторий в текущем каталоге и переключитесь на начальную ветвь проекта. Следующий шаг — установка зависимостей. В этой книге ради согласованности будем использовать менеджер npm (www.npmjs.com), но если вы предпочитаете программу yarn (другую библиотеку управления зависимостями, обертку для npm, yarn-pkg.com), работайте с ней. Нужно лишь убедиться, что вы устанавливаете yarn, а не npm.

Все модули, которые понадобятся для книги, должны быть включены в пакет .json в исходном коде приложения. Чтобы установить его, выполните следующую команду в каталоге исходного кода:

```
npm install
```

Команда установит все зависимости, которые вам понадобятся. Если вы измените версии узла (с помощью утилиты nvm или другими средствами), следует переустановить модули узла, потому что различные версии последнего будут компилировать модули по-разному (например, node-sass).

4.1.2. Какую версию узла следует использовать

Пришло время поговорить о том, какую версию узла предпочесть. Рекомендую последнюю стабильную версию. На момент написания книги это была линейка 8.X. Я не буду поддерживать версии узла ранее 6.X, так как рекомендуется поддерживать версии 8.X и выше, поскольку это не бизнес- или корпоративная среда, где вы не можете легко переключаться между версиями без тщательного тестирования. Узел 8.X также использует более новую версию npm и содержит значительные улучшения скорости, сделанные для базового движка V8.

Если на вашем компьютере нет ни одной из этих версий узла, перейдите на сайт nodejs.org, чтобы загрузить копию последней стабильной версии. Другой вариант — применить консольный инструмент nvm для установки локальных копий узла и возможности переключения между ними. Загружается инструмент nvm на странице github.com/creationix/nvm.

Различные версии узлов поддерживают разные возможности JavaScript, поэтому важно знать, что поддерживает ваша версия. Если хотите больше узнать о том, какие функции поддерживаются у вас и какие версии станут поддерживать другие, перейдите на сайт node.green, чтобы увидеть реализацию функций в разных версиях.

4.1.3. Замечание по инструментам и CSS

Как я упоминал в других книгах и разделах этой книги, инструментарий для JavaScript-приложений может быть сложным и быстро меняющимся от версии к версии. Это также домен, для которого характерны собственные обновления. По этим причинам я не буду рассказывать, как настраивать такие инструменты, как Webpack, Babel и пр. Для исходного кода приложения существует процесс разработки и сборки в правильном порядке, и вы можете исследовать конфигурацию, которую я настроил, но это выходит за рамки данной книги, поэтому я не буду ее описывать.

Еще один момент, который стоит отметить, — это каскадные таблицы стилей (CSS). Я уже рассмотрел способы работы с встроенными стилями в React, но обучение CSS также выходит за рамки этой книги. По этой причине я создал все стили, которые вам понадобятся. Любая разметка пользовательского интерфейса, которую вы видите, имеет созданные для нее стили. Некоторые стили зависят от определенных типов или иерархий, поэтому, если вы перемещаете разные элементы или меняете имена классов CSS, приложение может выглядеть искаженным. Моя цель — сделать так, чтобы вы меньше отвлекались, изучая React, но если хотите поэкспериментировать со стилями, я не буду вас ограничивать.

4.1.4. Развертывание

Приложение, запущенное на сайте social.react.sh, развернуто на сервере zeit.co, но если по какой-то причине в будущем возникнут обстоятельства, требующие изменения, я стану поддерживать работу приложения на любом облачном сервере, наиболее подходящем на тот момент. Вам не нужно беспокоиться о том, где размещено при-

ложение. Если в конце книги вы захотите развить и добавить приложение ради приобретения практики и получения удовольствия, вам нужно будет выбрать лучший способ развертывания. К счастью, процессы сборки и исполнения просты, так что найти сервер для развертывания будет довольно просто.

4.1.5. Сервер API и база данных

Чтобы не запускать базу данных, такую как MongoDB или PostgreSQL, мы воспользуемся имитацией REST API через библиотеку `JSON-server` (github.com/typicode/json-server). Я внес в настройки по умолчанию сервера некоторые изменения (вы найдете их в папке `db` репозитория), позволяющие немного упростить проект. Не выполняя сложной работы, вы получите простую базу данных, действия с которыми заключаются в чтении и изменении JSON-файла. Чтобы создать образец данных или сбросить данные приложения, можете выполнить следующую команду:

```
npm run db:seed
```

Она перепишет существующую базу данных JSON и заменит ее новыми образцами данных (пользователи, сообщения и комментарии — в духе франшизы «*Звездных войн*», да пребудет с вами сила). В последующих главах вы научитесь создавать пользователей в базе данных после авторизации. Если вы повторно запустите команду `seed` базы данных, созданный пользователь будет перезаписан и вам придется выйти из системы и снова авторизоваться, чтобы внести исправления. Так не должно быть, и вам, вероятно, не нужно будет запускать команду базы данных более одного раза. Но на всякий случай вы должны знать, что означает сброс данных.

Я задействовал ряд помощников, чтобы упростить выполнение запросов к API. Вы можете увидеть эти функции в файле `src/shared/http.js`. Я работаю с библиотекой `isomorphic-fetch` (github.com/matthew-andrews/isomorphic-fetch), потому что она дублирует стандартный API Fetch, доступный в браузерах, но работающий на сервере. Я предполагаю, что у вас есть некоторый опыт работы с библиотекой HTTP в браузере, но если его нет, используйте включенные файлы помощников как способ начать изучение API Fetch (developer.mozilla.org/ru/docs/Web/API/Fetch_API).

4.1.6. Запуск приложения

Самый простой способ запустить приложение в режиме разработки — выполнить следующую команду:

```
npm run dev
```

Существуют и другие команды, но главная, которая вам понадобится, — это `dev`. Чтобы просмотреть другие доступные команды, можете выполнить такую команду:

```
npm run
```

Она отобразит каждую доступную команду для репозитория. Пробуйте каждую из них, чтобы увидеть, для чего они используются. Две главные команды, которые вам понадобятся, — `npm run dev` и `npm run db:seed`.

4.2. Процесс рендеринга и методы жизненного цикла

Если вы клонировали репозиторий и установили зависимости, значит, у вас есть все, что нужно. Прежде чем начать создавать приложение Letters Social, рассмотрите рендеринг и методы жизненного цикла. Это ключевые особенности React, и, разобравшись с ними, вы будете лучше подготовлены к созданию приложения Letters Social.

4.2.1. Знакомство с методами жизненного цикла

В главе 2 мы продемонстрировали, что возможно создавать и назначать функции в качестве обработчиков событий (щелчков кнопкой мыши, отправки формы и т. д.) внутри своих компонентов. Это полезно, потому что вы можете создавать динамические компоненты, реагирующие на пользовательские события (ключевая характеристика любого современного веб-приложения). Но что, если вы хотите чего-то большего? Если у вас есть только эта функция, вам кажется, что вы все еще работаете со старым добрым кодом HTML и JavaScript. Предположим, вы хотите получить данные пользователя из API или прочитать cookie-файлы для последующего применения, не ожидая инициированного пользователем события. Это обычные действия, которые нужно выполнять в веб-приложениях, — в некоторых случаях придется делать это автоматически, так где же это должно происходить? Ответ — в методах жизненного цикла.

ОПРЕДЕЛЕНИЕ

Методы жизненного цикла — это специальные методы, связанные с компонентами, основанными на классе React, которые будут выполняться в определенных точках жизненного цикла компонента. Жизненный цикл — это способ отразить процесс существования компонента. Компонент с жизненным циклом имеет метафорическую «жизнь» — по крайней мере начало, середину и конец. Эта ментальная модель упрощает его понимание и показывает, на каком этапе своей жизни он находится. Методы жизненного цикла React не уникальны — многие технологии пользовательского интерфейса используют их из-за интуитивного характера и приносимой пользы. Основными частями жизни компонента React являются монтирование, обновление и размонтирование. На рис. 4.2 представлен обзор жизненного цикла компонента и процесса рендеринга (то, как React управляет компонентами с течением времени).

Я упоминал методы жизненного цикла в предыдущих главах, но теперь пришло время действительно углубиться в них, чтобы понять, что они собой представляют

и как их применять. Для начала вновь окиньте взглядом React. Посмотрите на верхнюю часть рис. 4.2, чтобы освежить память. Я говорил о состоянии в React, создавая компоненты с помощью функции `React.create-Element` и синтаксического сахара JSX, но нам все равно нужно подробно изучить методы жизненного цикла.

Рис. 4.2. Обзор React. React будет отображать компоненты (создавать их, управлять ими) и формировать из них пользовательские интерфейсы

Освежим знания из прошлых глав и рассмотрим некоторые понятия. Что такое рендеринг (или визуализация)? Одно из определений *рендеринга* — «привести в состояние». В рамках изучаемой темы можете представить рендеринг как процесс библиотеки React по созданию пользовательского интерфейса и управлению им. Это работа по выводу приложения на экран. React берет ваши компоненты и превращает их в пользовательский интерфейс.

Подключайтесь к этому процессу, используя методы жизненного цикла, о которых узнали в данной главе. Они обеспечивают гибкие способы нужного воздействия в определенный момент жизни компонента и доступны только для компонентов, созданных из классов, расширяющих абстрактный базовый класс `React.Component`.

Функциональные компоненты без состояния, описанные в конце главы 3, не имеют доступных методов жизненного цикла. У вас также не получится задействовать метод `this.setState` внутри них, потому что у них нет экземпляров поддержки: React не отслеживает какое-либо внутреннее состояние для них. Их данные все еще способен обновить родитель свойства, но вы не получите доступа к методам жизненного цикла. Это может показаться проблемой или ограничением их функциональности, но во многих случаях вам больше ничего не нужно.

4.2.2. Типы методов жизненного цикла

В этом подразделе рассматриваются различные методы жизненного цикла, предоставляемые React в разных группах, и обсуждаются их функции. Методы жизненного цикла разделяют на две основные группы:

- *методы типа «будет»* — вызываются прямо перед тем, как что-то произойдет;
- *методы типа «было»* — вызываются сразу после того, как что-то произошло.

Существует также еще несколько методов, которые не вписываются ни в одну из этих групп, они связаны с инициализацией и обработкой ошибок, и один — для обновления. Однако большинство методов относятся к типам «было» или «будет».

Мы можем разложить их еще на несколько типов, исходя из того, с какой частью жизненного цикла они связаны (рис. 4.3). Каждый компонент имеет соответствующие методы жизненного цикла. Последний делится на четыре основные части:

- *инициализация* — при создании экземпляра класса компонента;
- *монтирование* — компонент вставляется в DOM;
- *обновление* — компонент обновляется новыми данными через состояние или свойства;
- *размонтирование* — компонент удаляется из DOM.

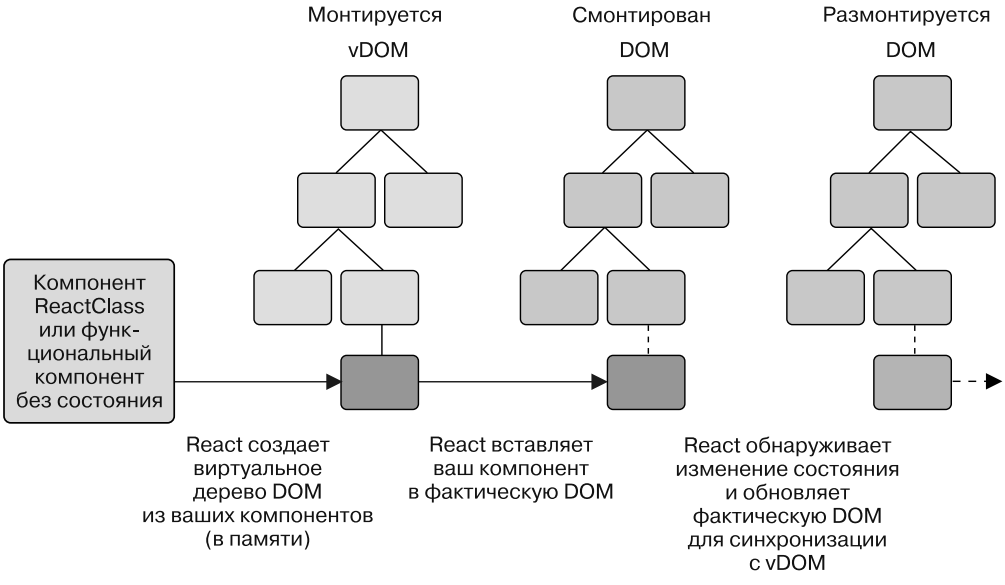


Рис. 4.3. Обзор процесса рендеринга и жизненного цикла компонента. Это процесс, который использует React, чтобы управлять вашими компонентами самостоятельно. Три основные части жизни компонента: он монтируется, смонтирован и размонтируется. Монтируется, когда помещается в DOM, смонтирован, как только оказался внутри модели, и размонтируется при удалении

Существуют методы жизненного цикла, которые будут вызываться во время инициализации, а также до и после монтирования, обновления и размонтирования компонентов. Этих методов не так много, особенно по сравнению с другими библиотеками и фреймворками, но их легко перепутать, когда вы изучаете React. Формирование значимых ментальных групп для них поможет вам ориентироваться в разных частях процесса рендеринга. На рис. 4.4 представлен обзор всего процесса рендеринга в React, который мы рассмотрим более подробно в ходе этой главы.

Помните, что размышления о пользовательских интерфейсах и компонентах в терминах жизненного цикла неуникальны для React или JavaScript. Другие технологии восприняли эту идею с большим энтузиазмом, иногда их на это вдохновила React (см., например, componentkit.org). Но эти конкретные методы жизненного цикла *уникальны* для React. Чтобы изучить их, вы создадите два простых компонента: родительский и дочерний, которые будут реализовывать все методы жизненного цикла, которые мы рассмотрим. Перейдите на страницу codesandbox.io/s/2vxn9251xy, чтобы узнать, как добавить эти компоненты. Вы также можете скачать код с сайта CodeSandbox и использовать инструменты разработчика своего браузера для проверки данных в консоли. В листинге 4.1 показана базовая настройка этих компонентов.

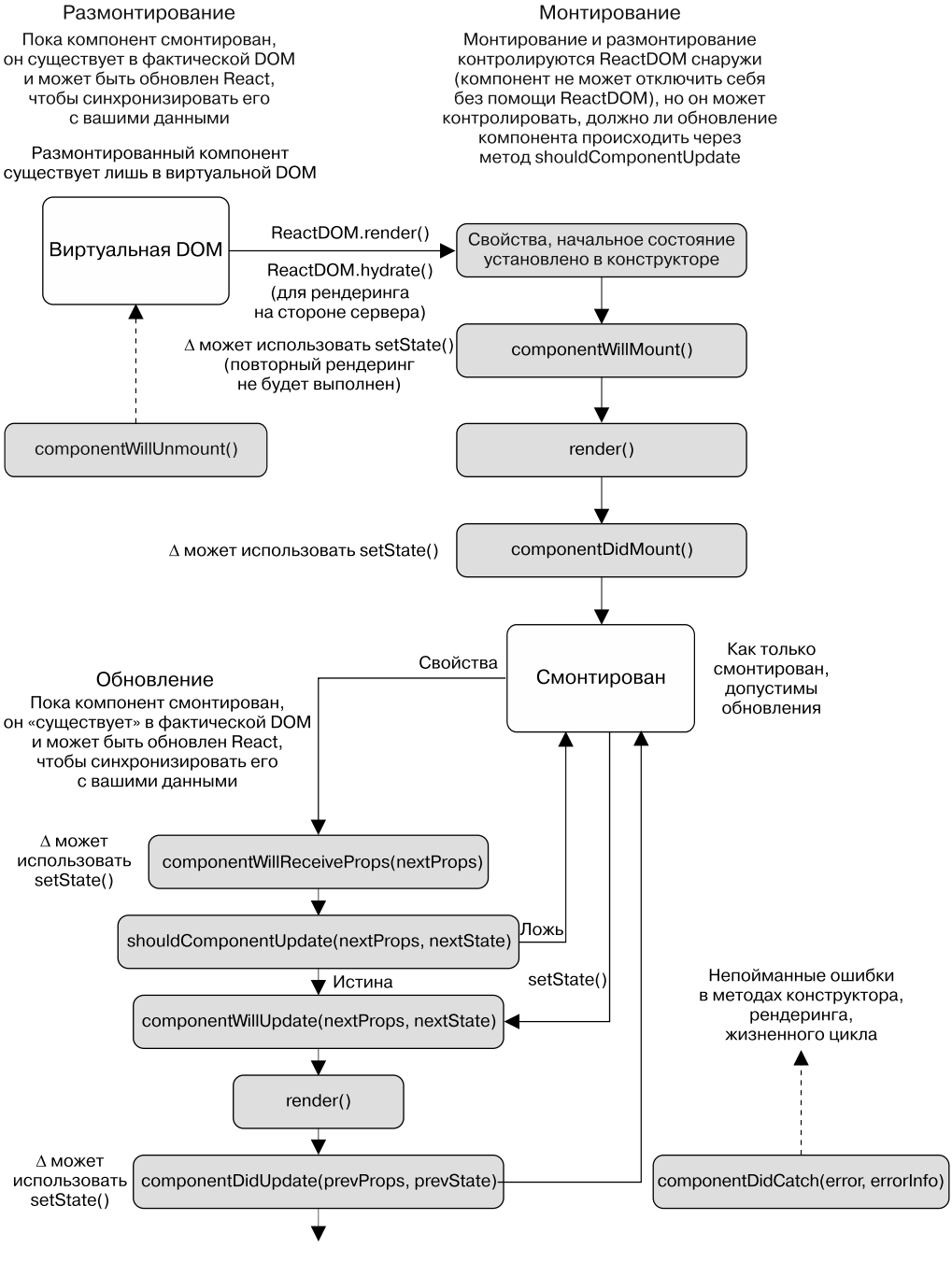


Рис. 4.4. Обзор жизненного цикла компонента в React. ReactDOM отображает компонент, и определенные методы жизненного цикла вызываются в процессе управления им

Листинг 4.1. Изучение методов жизненного цикла

```
import PropTypes from 'prop-types';
import React, { Component } from 'react';
import { render } from 'react-dom';

class ChildComponent extends Component {
  static propTypes = {
    name: PropTypes.string
  };
  static defaultProps = (function() {
    console.log('ChildComponent : defaultProps');
    return {};
  })();
  constructor(props) {
    super(props);
    console.log('ChildComponent: state');
  }
  render() {
    console.log('ChildComponent: render');
    return (
      <div>
        Name: {this.props.name}
      </div>
    );
  }
};

class ParentComponent extends Component {
  constructor() {
    super(props);
    this.state = {
      name: ''
    }
    this.onInputChange =
          this.onInputChange.bind(this);
  }
  onInputChange(e) {
    this.setState({ text: e.target.value });
  }
  render() {
    console.log('ParentComponent: render');
    return [
      <h2 key="h2">Learn about rendering and lifecycle methods!</h2>,
      <input key="input" value={this.state.text}
      onChange={this.onInputChange} />,
      <ChildComponent key="ChildComponent" name={this.state.text} />
    ];
  }
};

render(
```

Объявление дочернего компонента

Установка propTypes в виде статического метода для класса

Установка свойства по умолчанию — обычно вы должны установить объект, а не функцию, но вы сразу используете исполняющую функцию для ввода инструкции console.log

Создание родительского компонента

Привязка метода onInputChange в конструкторе, чтобы вы могли сослаться на метод в функции render и указать экземпляр класса, а не определение

Обновление состояния с помощью данных из формы

Отображение дочернего компонента внутри родителя

```
  <ParentComponent />,
  document.getElementById('container')
);
```

Использование React DOM для рендеринга родительского компонента

Изучая методы жизненного цикла, не нужно требовать от компонентов множества функций. В примере вы настроили родителя и потомка. Родительский компонент прослушивает изменения в поле ввода и предоставляет новые свойства дочернему через состояние.

4.2.3. Начальный метод и метод типа «будет»

Первой группой свойств, связанных с жизненным циклом, являются начальные свойства компонента. К ним относятся два, о которых вы уже знаете: `defaultProps` и `state` (начальное). Они помогают предоставить исходные данные вашему компоненту. Взглянем на них, прежде чем читать дальше.

- `defaultProps` — статическое свойство, которое определяет свойства по умолчанию для компонента. Устанавливает `this.props`, если это свойство не задано родительским компонентом, открывается до того, как какие-либо компоненты смонтированы, и не может полагаться на `this.props` или `this.state`. Поскольку `defaultProps` является статическим свойством, оно доступно из класса, а не из экземпляров.
- `state` *(начальное)* — значение этого свойства в конструкторе будет начальным, заданным для определения состояния вашего компонента. Это особенно полезно, когда нужно обеспечить содержимое заполнителя, установить значения по умолчанию и т. п. Оно похоже на свойства по умолчанию, за исключением того, что данные должны быть изменены и доступны только для компонентов, которые наследуются от `React.Component`.

Несмотря на то что начальное состояние и свойства не задаются специальными методами из класса `Component` React (они используют JavaScript-метод `constructor`), они все же являются частью жизненного цикла компонента. Вы можете забыть о них, но они играют важную роль в предоставлении данных для ваших компонентов.

Чтобы проиллюстрировать порядок рендеринга и различные методы жизненного цикла, которые мы будем рассматривать, вы создадите два простых компонента, для которых сможете указать методы жизненного цикла. Вы создадите родительский и дочерний компоненты, чтобы увидеть не только порядок, в котором вызываются различные методы, но и то, каков он между родителями и потомками. Для упрощения вы будете обрабатывать информацию только в консоли разработчика. На рис. 4.5 показано, что вы увидите на панели инструментов разработчика, когда закончите работу.

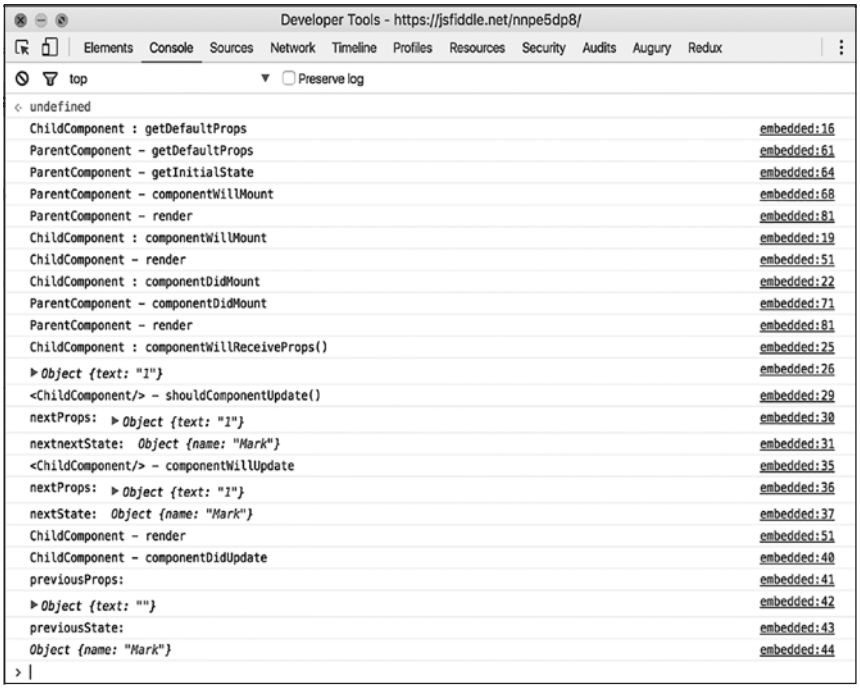


Рис. 4.5. Вывод обрабатываемых компонентов. Метод жизненного цикла инициирует вывод сообщения, которое регистрируется в консоли на каждом шаге, а также любые аргументы, доступные этому методу. Методы жизненного цикла в действии можно увидеть на странице codesandbox.io/s/2vxn9251xy

4.2.4. Монтирование компонентов

Теперь, когда вы создали родительский и дочерний компоненты, перейдем к монтированию. *Монтирование* — это процесс помещения компонента в DOM. Помните, что компоненты существуют только в виртуальной DOM, пока React не создаст их в фактической DOM. На рис. 4.6 представлен обзор процессов монтирования и рендеринга для родительского и дочернего компонентов. Способы монтирования позволят вам «войти» в начало и конец жизни компонента и будут запускаться только один раз, потому что у него по определению могут быть только одно начало и один конец.

ОПРЕДЕЛЕНИЕ

Монтирование — это процесс React, помещающий компоненты в фактическую DOM. После этого компонент «готов» и, как правило, наступает самое время выполнять HTTP-вызовы или чтение cookie-файлов. На этом этапе вы также можете получить доступ к DOM-элементу через функцию *ref*, обсуждаемую в следующих главах.

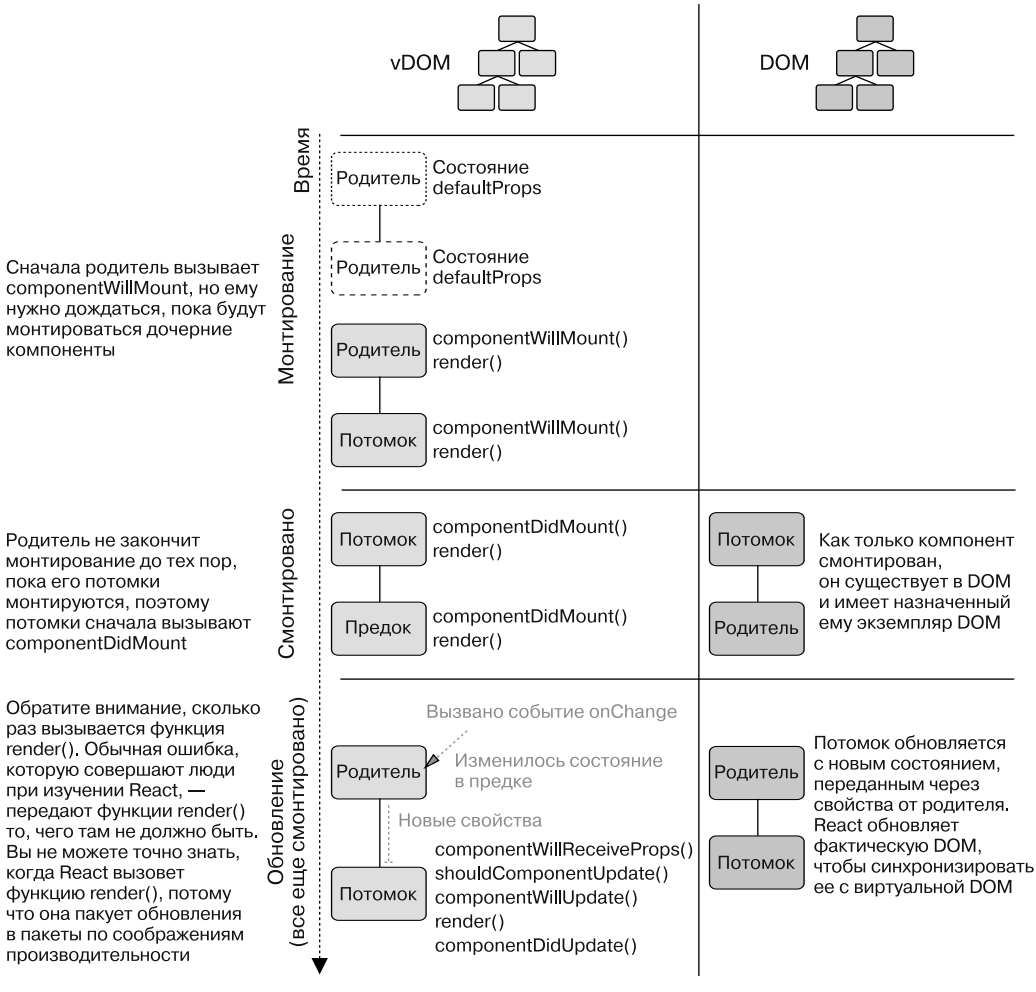


Рис. 4.6. Процесс рендеринга, применяемый к родительскому и дочернему компонентам

Если вы посмотрите на рис. 4.6, то заметите, что существует только одна возможность изменить состояние до того, как компонент будет смонтирован. Для этого используйте функцию `componentWillMount`, которая позволяет установить состояние или выполнить другие действия перед монтированием компонентов. Любые изменения состояния внутри этого метода не будут запускать повторный рендеринг, в отличие от других обновлений состояния, которые инициируют процесс обновления, рассмотренный ранее. Важно знать, какие методы запускают повторный рендеринг, а какие — нет. Так вы сможете понять, как будет вести себя приложение, а также отлаживать его, если что-то пойдет не так. На рис. 4.7 показаны методы монтирования в контексте обзора жизненного цикла, с которым мы работали ранее.

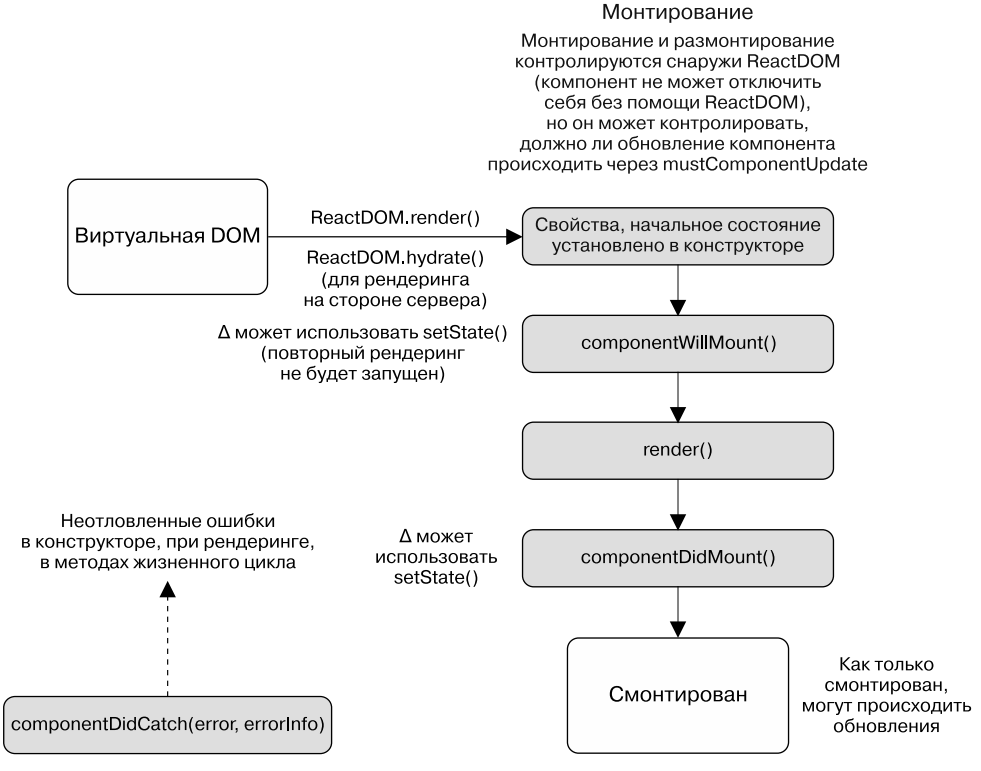


Рис. 4.7. Способы монтирования в контексте более крупного процесса жизненного цикла. Компоненты добавляются в DOM, и пока они там присутствуют, вызываются несколько специфических методов

Следующий метод, который я рассмотрю, — `componentDidMount`. Когда React вызывает этот метод, у вас есть возможность использовать его, а также получить доступ к ссылкам на компоненты. В этом методе у вас есть доступ к состоянию и свойствам компонентов, также вы узнаете, что компонент готов к обновлению. Теперь можно обновить его с помощью данных, возвращаемых в результате сетевого запроса. Отличный вариант для работы со сторонними библиотеками, зависимыми от DOM, такими как jQuery и др.

Если вы запустите обработчики или иные функции в других методах, таких как `render()`, то из-за специфики работы React получите непредсказуемые и неожиданные результаты. Методы рендеринга должны быть *чистыми* (согласованными на основе заданного ввода), обычно они вызываются несколько раз на протяжении жизни компонента. React способна даже собирать обновления в пакеты, поэтому нет гарантии, что рендеринг будет выполнен в конкретный момент времени.

Теперь, когда мы рассмотрели некоторые методы, связанные с монтированием, вы добавите их в свои компоненты, чтобы просмотреть их жизненный цикл. В листинге 4.2 показано, как это делается.

Листинг 4.2. Методы монтирования

```
import PropTypes from 'prop-types';
import React, { Component } from 'react';
import { render } from 'react-dom';

class ChildComponent extends Component {
  static propTypes = {
    name: PropTypes.string
  };
  static defaultProps = (function() {
    console.log('ChildComponent : defaultProps');
    return {};
  })();
  constructor(props) {
    super(props);
    console.log('ChildComponent: state');
    this.state = {
      name: 'Mark'
    };
  }
  componentWillMount() {
    console.log('ChildComponent : componentWillMount');
  }
  componentDidMount() {
    console.log('ChildComponent : componentDidMount');
  }
  render() {
    if (this.state.oops) {
      throw new Error('Something went wrong');
    }
    console.log('ChildComponent: render');
    return [
      <div key="name">Name: {this.props.name}</div>
    ];
  }
}

class ParentComponent extends Component {
  static defaultProps = (function() {
    console.log('ParentComponent: defaultProps');
    return {
      true: false
    };
  })();
  constructor(props) {
    super(props);
    console.log('ParentComponent: state');
    this.state = { text: '' };
    this.onInputChange = this.onInputChange.bind(this);
  }
  componentWillMount() {
    console.log('ParentComponent: componentWillMount');
  }
  componentDidMount() {
    console.log('ParentComponent: componentDidMount');
  }
```

Добавление функций componentDidMount и componentWillMount в дочерний компонент

Добавление функций componentDidMount и componentWillMount в родительский компонент

```
  onInputChange(e) {
    const text = e.target.value;
    this.setState(() => ({ text: text }));
  }
  render() {
    console.log('ParentComponent: render');
    return [
      <h2 key="h2">Learn about rendering and lifecycle methods!</h2>,
      <input key="input" value={this.state.text}
  onChange={this.onInputChange} />,
      <ChildComponent key="ChildComponent" name={this.state.text} />
    ];
  }
}

render(<ParentComponent />, document.getElementById('root'));
```

Упражнение 4.1. Размышляя о монтировании

Что означает выражение «компонент *смонтирован*»?

4.2.5. Методы обновления

Итак, компонент смонтирован и находится и в DOM, теперь его можно обновить. В главе 3 вы видели, что функция `this.setState()` применяется для выполнения мелкого слияния новых данных и состояния компонента, но при запуске обновления происходит нечто большее. React предоставляет методы `shouldComponentUpdate`, `componentWillUpdate` и `componentDidUpdate`, которые используются для подключения к процессу обновления. На рис. 4.8 показан фрагмент процесса обновления из общей диаграммы жизненного цикла, которую мы рассмотрели ранее.

В отличие от других методов, которые вы видели до сих пор, здесь есть возможность контролировать, должно ли происходить обновление. Другое различие между методами обновления и теми, которые связаны с монтированием, заключается в том, что они предоставляют аргументы для свойств и состояния. С их помощью определяют, должно ли происходить обновление, реагируют на изменения.

Если метод `shouldComponentUpdate` по какой-либо причине возвращает `false`, вызов функции `render()` пропускается до следующего изменения состояния. Это означает, что вы можете предотвратить ненужное обновление компонента. Поскольку он не будет обновлен, методы `componentWillUpdate` и `componentDidUpdate` вызываться не будут.

Если вы не укажете иное, метод `shouldComponentUpdate` будет всегда возвращать `true`, но если всегда будете осторожно относиться к состоянию как неизменяемому и производить чтение только из свойств и состояния в функции `render()`, то сможете переопределить метод `shouldComponentUpdate` с реализацией, которая сравнивает старые свойства и состояние с их заменой. Это полезно для улучшения производительности, но только в крайнем случае. React уже использует сложные и продвинутые методы для определения того, что нужно обновлять и когда.

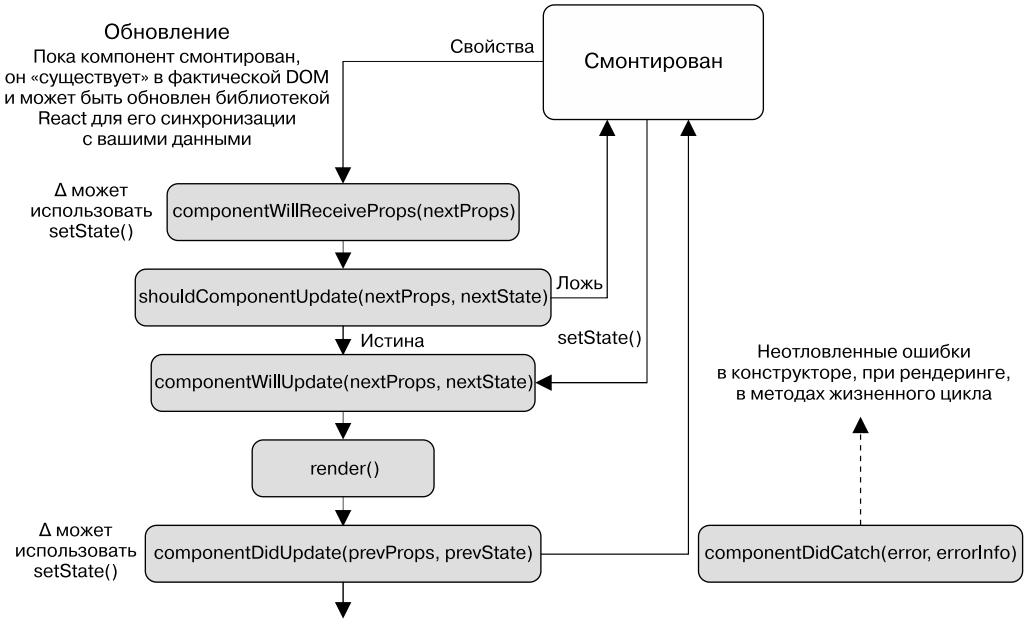


Рис. 4.8. Обновление методов жизненного цикла. Когда компонент обновляется, запускаются несколько перехватчиков, позволяющих определить, должен ли он обновляться вообще, как это делать и когда

Если вы все же используете метод `shouldComponentUpdate`, значит, описанные ранее методы по какой-либо причине применить нельзя. Это не значит, что вы никогда не должны задействовать метод `shouldComponentUpdate`, но он, скорее всего, не понадобится, когда вы начнете работу с React. Как и все методы жизненного цикла, он доступен, но работать с ним нужно только в случае необходимости. В листинге 4.3 показан пример связанных с обновлением жизненных циклов React.

Теперь, когда вы указали методы обновления для своих компонентов, попробуйте запустить их снова и введите что-нибудь в текстовое поле. Вы увидите каскадный вывод в консоли разработчика (в листинге 4.4 показано, какие компоненты должны выводиться). Потратьте минутку, чтобы посмотреть на порядок рендеринга. Заметили что-нибудь? Порядок должен быть тем же, о котором вы узнали из этой главы, но теперь вы видите, как упорядочиваются дочерние и родительские компоненты. Как говорилось в главе 2, библиотека React рекурсивна в том, как она формирует дерево и рендерит объекты, — она всесторонне изучает все части ваших компонентов, спрашивая о каждом из них и обо всех их дочерних элементах.

Зная все, что нужно знать о вашем дереве компонентов, React может автоматически грамотно создавать компоненты в правильном порядке. Из листинга 4.4 видно, что дочерний компонент монтируется перед родительским. Это имеет смысл при установке родителя: потомки должны быть созданы до того, как она будет считаться завершенной. Если потомка еще не существует, нельзя сказать, что родительский элемент был смонтирован.

Листинг 4.3. Методы обновления

```
//...
class ChildComponent extends Component {
  //...
  componentWillReceiveProps(nextProps) {

    console.log('ChildComponent : componentWillReceiveProps()');
    console.log('nextProps: ', nextProps);
  }
  shouldComponentUpdate(nextProps, nextState) {
    console.log('<ChildComponent/> - shouldComponentUpdate()');
    console.log('nextProps: ', nextProps);
    console.log('nextState: ', nextState);
    return true;
  }
  componentWillUpdate(nextProps, nextState) {
    console.log('<ChildComponent/> - componentWillUpdate');
    console.log('nextProps: ', nextProps);
    console.log('nextState: ', nextState);
  }
  componentDidUpdate(previousProps, previousState) {
    console.log('ChildComponent: componentDidUpdate');
    console.log('previousProps:', previousProps);
    console.log('previousState:', previousState);
  }
  //...
  render() {
    console.log('ChildComponent: render');
    return [
      <div key="name">Name: {this.props.name}</div>
    ];
  }
}

class ParentComponent extends Component {
  //...
  onInputChange(e) {
    const text = e.target.value;
    this.setState(() => ({ text: text }));
  }
  //...
  render() {
    console.log('ParentComponent: render');
    return [
      <h2 key="h2">Learn about rendering and lifecycle methods!</h2>,
      <input key="input" value={this.state.text}
      onChange={this.onInputChange} />,
      <ChildComponent key="ChildComponent" name={this.state.text} />
    ];
  }
}
//...
```

Добавление методов обновления к дочернему компоненту, чтобы проверить процесс обновления на одном компоненте

Добавление методов обновления к дочернему компоненту, чтобы проверить процесс обновления на одном компоненте

Листинг 4.4. Вывод обновления компонента с введенным текстом

```
ChildComponent : defaultProps
ParentComponent : defaultProps
ParentComponent : get initial State
ParentComponent : componentWillMount
ParentComponent : render
ChildComponent : componentWillMount
ChildComponent : render
ChildComponent : componentDidMount
ParentComponent : componentDidMount
ParentComponent : render
ChildComponent : componentWillReceiveProps
Object {text: "Mark"}
<ChildComponent/> : shouldComponentUpdate
nextProps:  Object {text: "Mark"}
nextnextState:  Object {name: "Mark"}
<ChildComponent/> : componentWillUpdate
  nextProps:  Object {text: "Mark"}
  nextState:  Object {name: "Mark"}
  ChildComponent : render
  ChildComponent : componentDidUpdate
  previousProps:  Object {text: ""}
  previousState:  Object {name: "Mark"}
>
```

Слово Mark было добавлено, поэтому вы не инициируете серию обновлений для каждой буквы

Кроме того, вы заметите, что, когда происходит обновление, дочерний компонент получает свойства, потому что свойство этого дочернего элемента было изменено родителем с помощью функции `this.setState()`. С этого времени методы обновления выполняются в следующем порядке: `shouldComponentUpdate`, `componentWillUpdate`, `componentDidUpdate`. Если вы указали, что компонент по каким-то причинам не должен обновляться, вернув значение `false` метода `mustComponentUpdate`, эти шаги будут пропущены.

4.2.6. Методы размонтирования

Так же как мы могли отслеживать монтирование компонента, мы можем прослушивать его размонтирование. *Размонтирование* — это процесс удаления компонента из DOM. Если ваше приложение написано с помощью React, *роутер* (рассмотрим в главах 8 и 9) удалит компоненты при перемещении по страницам. Но реально интегрировать React для работы с другими фреймворками и библиотеками, поэтому вам может потребоваться выполнить некоторые другие действия, когда ваш компонент размонтируется (например, очистка интервала, переключение настройки и т. д.). Независимо от действия пользуйтесь методом `componentWillUnmount`, чтобы выполнить любую очистку, необходимую для удаления компонента. На рис. 4.9 показано, как происходит размонтирование.

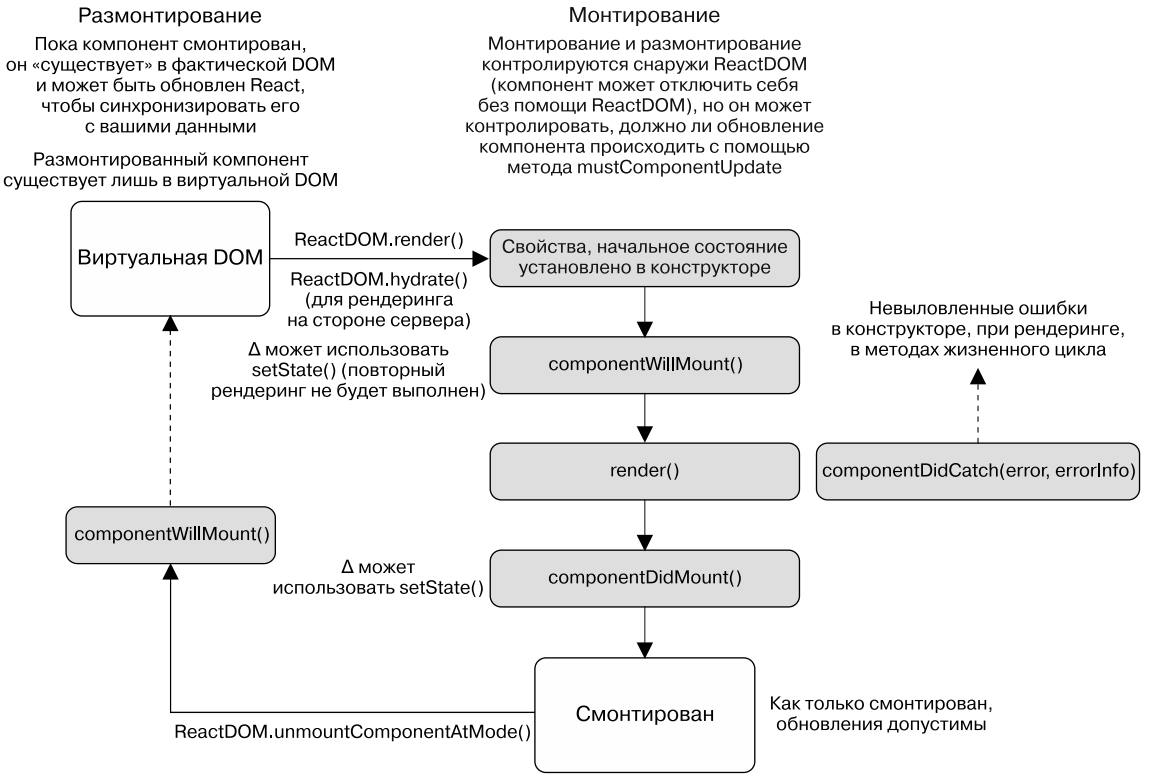


Рис. 4.9. React DOM отвечает за монтирование и размонтирование компонентов. Монтирование — это процесс вставки ваших компонентов в DOM, а размонтирование — обратное действие — их удаление из DOM. Когда компоненты размонтированы, они больше не существуют в DOM

Основываясь на том, как монтирование работало до сих пор, вы можете ожидать, что будет доступен метод `componentDidUnmount`, но вы заблуждаетесь. Это потому, что, как только компонент удаляется, его жизнь заканчивается и «воскрешению» он не подлежит. Добавим метод `componentWillUnmount` в наш пример, чтобы получить полную картину жизненного цикла компонента (листинг 4.5).

Листинг 4.5. Размонтирование

Добавление метода componentWillUnmount к родительскому и дочернему компонентам

```
//...
class ChildComponent extends Component {
  //...
  componentWillUnmount() {
    console.log('ChildComponent: componentWillUnmount');
  }
  render() {
    console.log('ChildComponent: render');
    return [
      <div key="name">Name: {this.props.name}</div>
    ];
  }
}

class ParentComponent extends Component {
  //...
  componentWillUnmount() {
    console.log('ParentComponent: componentWillUnmount');
  }
  onInputChange(e) {
    const text = e.target.value;
    this.setState(() => ({ text: text }));
  }
  componentDidCatch(err, errorInfo) {
    console.log('componentDidCatch');
    console.error(err);
    console.error(errorInfo);
    this.setState(() => ({ err, errorInfo }));
  }
  render() {
    return [
      <h2 key="h2">Learn about rendering and lifecycle methods!</h2>,
      <input key="input" value={this.state.text}
  onChange={this.onInputChange} />,
      <ChildComponent key="ChildComponent" name={this.state.text} />
    ];
  }
}
//...
```

4.2.7. Перехват ошибок

Обработка ошибок — это отличный шаг к разработке чистых программ. До сих пор мы не использовали каких-либо специальных методов в React для работы с ошибками. Если вы долго работали с React, то наверняка помните, что предыдущие версии React блокировали приложение целиком, если возникла ошибка в методах `render` или жизненного цикла компонента React. Это грустно, так как означает, что неперехваченная ошибка может заблокировать работу всего приложения.

В более поздних версиях React была введена новая концепция, называемая *границами ошибок*. Если неперехваченное исключение выброшено внутри функций `constructor`, `render` или методов жизненного цикла компонента, React размонтирует компонент и его потомков из DOM. Сначала это способно запутать, но преимущество, которое предлагает такой подход, — это возможность изолировать ошибки в компонентах от сбоев остальной части приложения.

Упражнение 4.2. Различия между компонентами

Какие вы знаете различия между компонентами React, созданными из абстрактного базового класса `React.Component`, и компонентами, созданными из простых функций без наследования?

Вы можете обрабатывать эти ошибки с помощью другого метода, который компоненты наследуют от `React.Component`, — `componentDidCatch`. Синтаксис метода похож на конструкцию `try...catch`, которую вы встречаете в JavaScript-сценариях. Метод `componentDidCatch` предоставляет доступ к выброшенной ошибке и сообщению об ошибке. Используя их, вы обеспечите правильное реагирование компонентов на ошибки. В более сложном приложении этот метод можно применять для настройки состояния ошибки для отдельных компонентов (например, виджета, карты или другого компонента) или на уровне приложения. В листинге 4.6 показано, как добавить метод `componentDidCatch` к родительскому компоненту.

Листинг 4.6. Обработка ошибок

```
//...
class ChildComponent extends Component {
  constructor(props) {
    super(props);
    console.log('ChildComponent: state');
    this.oops = this.oops.bind(this);
  }
  //...
  oops() {
    this.setState(() => ({ oops: true }));
  }
```

Связывание метода класса

Переключение состояния для выброса ошибки

```
  render() {
    console.log('ChildComponent: render');
    if (this.state.oops) {
      throw new Error('Something went wrong');
    }
    return [
      <div key="name">Name: {this.props.name}</div>,
      <button key="error" onClick={this.oops}>
        Create error
      </button>
    ];
  }
}

class ParentComponent extends Component {
  //...
  constructor(props) {
    super(props);
    console.log('ParentComponent: state');
    this.state = { text: '' };
    this.onInputChange = this.onInputChange.bind(this);
  }
  //...
  componentDidCatch(err, errorInfo) {
    console.log('componentDidCatch');
    console.error(err);
    console.error(errorInfo);
    this.setState(() => ({ err, errorInfo }));
  }
  render() {
    console.log('ParentComponent: render');
    if (this.state.err) {
      return (
        <details style={{ whiteSpace: 'pre-wrap' }}>
          {this.state.error && this.state.error.toString()}
          <br />
          {this.state.errorInfo.componentStack}
        </details>
      );
    }
    return [
      <h2 key="h2">Learn about rendering and lifecycle methods!</h2>,
      <input key="input" value={this.state.text}
          onChange={this.onInputChange} />,
      <ChildComponent key="ChildComponent" name={this.state.text} />
    ];
  }
}

render(<ParentComponent />, document.getElementById('root'));
```

Выброс ошибки в методе render

Добавление метода componentDidCatch родителю и использование его для обновления состояния компонента

Если ошибка выброшена, отображение ошибки и сообщения об ошибке

Мы рассмотрели различные методы жизненного цикла, предоставляемые React, и изучили, как работать с ними в различных ситуациях. Если вам кажется, что методов слишком много, то вам будет приятно узнать, что они составляют основную часть API для компонентов React (можете пользоваться табл. 4.1 как памяткой). Основной API библиотеки React содержит не намного больше методов, чем рассмотрено на данный момент. Более того, вам не нужно использовать каждый из этих методов — применяйте только необходимые. В табл. 4.1 приведен обзор рассмотренных методов (обратите внимание на то, что функции `render` среди них нет).

Таблица 4.1. Сводка методов жизненного цикла компонента React

Метод	Начальный	Будет	Было
Монтирование	defaultProps. Аргументы — нет, статическое свойство. Что — статическая версия доступна многократно. Устанавливает значения в this.props, если этот параметр не задан родительским компонентом. Когда — вызывается, если компонент создан и не может полагаться на this.props. Возвращенные сложные объекты совместно используются экземплярами, а не копируются	componentWillMount. Аргументы — нет. Что — позволяет управлять данными компонентов до начала процесса монтирования. Например, если вы вызываете setState в этом методе, функция render() будет видеть обновленное состояние и выполнится только один раз, несмотря на изменение состояния. Последний шанс изменить исходные данные для рендеринга. Когда — вызывается однократно как с клиентской, так и с серверной стороны (в главе 12 описывается рендеринг на стороне сервера) непосредственно перед началом рендеринга	componentDidMount. Аргументы — нет. Что — вызывается, как только компонент был помещен в DOM. На этом этапе вы получаете доступ к ссылкам (способ доступа к базовому представлению DOM, обсуждаемому в последующих главах). Удобная позиция для выполнения «нечистых» действий, таких как интеграция с другими библиотеками JavaScript, установка таймеров (через setTimeout или setInterval) или отправка HTTP-запросов. Мы часто будем использовать этот метод для замены данных-заполнителей в компонентах. Когда — вызывается однократно, только со стороны клиента (не на сервере!), сразу после первоначального рендеринга. Метод componentDidMount() дочерних компонентов вызывается перед родительскими компонентами

Продолжение ⇩

Таблица 4.1 *(продолжение)*

Метод	Начальный	Будет	Было
Обновление	shouldComponentUpdate. Аргументы — nextProps, nextState. Что — если shouldComponentUpdate возвращает false, то функция render() будет полностью пропущена до следующего изменения состояния. Кроме того, методы componentWillUpdate и componentDidUpdate не будут вызываться. Полезно как запасной выход для расширенной настройки производительности. Когда — вызывается перед рендерингом, когда ваш компонент принимает новые свойства или состояние. Не вызывается для первоначального рендеринга	componentWillReceiveProps. Аргументы — nextProps: Object. Что — используйте как возможность реагировать на передачу свойства, прежде чем функция render() вызывается обновлением состояния с помощью метода this.setState(). Доступ к старым свойствам можно получить через this.props. Вызов метода this.setState() внутри этой функции не вызовет дополнительный рендеринг. Когда — вызывается, когда компонент получает новые свойства. Этот метод не вызывается для первоначального рендеринга. componentWillUpdate. Аргументы — nextProps: Object, nextState: Object. Что — используйте в качестве возможности выполнить подготовку до того, как произойдет обновление. Вам нельзя применять функцию setState(). Когда — вызывается непосредственно перед рендерингом при получении новых свойств или состояния. Не вызывается для первоначального рендеринга. componentWillUnmount. Аргументы — нет. Что — выполните в этом методе любую необходимую очистку, например обнуление таймеров или очистку любых DOM-элементов, созданных методом componentDidMount. Когда — вызывается непосредственно перед размонтированием компонента	componentDidUpdate. Аргументы — prevProps: Object, prevState: Object. Что — вызывается сразу после того, как обновления компонентов включены в DOM. Этот метод не вызывается для первоначального рендеринга. Когда — используйте в качестве возможности работать с DOM после того, как компонент был обновлен
Ошибки	componentDidCatch. Аргументы — error, errorInfo. Что — обрабатывает ошибки в компонентах. React будет отключать компоненты в дереве ниже позиции, где произошла ошибка. Когда — вызывается при ошибке в конструкторе, методах жизненного цикла или рендеринге		

4.3. Начало создания Letters Social

Теперь, когда вы немного больше знаете о методах жизненного цикла React и о том, что с их помощью можно сделать, давайте реализовывать полученные знания. Приступим к разработке приложения Letters Social. Если вы еще этого не сделали, прочитайте раздел 4.1, чтобы узнать, как использовать репозиторий Letters Social. Сначала вы находились в первой ветви, но, добравшись до конца главы, окажетесь в ветке главы 4 (`git checkout chapter-4`).

До этого момента вы выполняли бóльшую часть своего кода на сайте CodeSandbox в браузере. Это было удобно для обучения, но теперь займемся редактированием файлов на локальном компьютере. Вы захотите поручить процесс сборки инструменту Webpack, включенному в репозиторий, по нескольким причинам. Он позволяет:

- писать JavaScript-код во многие файлы вывода с автоматической установкой зависимостей и порядка импорта;
- поддерживать и обрабатывать файлы разных типов (например, SCSS- или файлы шрифтов);
- использовать другие инструменты сборки, такие как Babel, чтобы работать с современной версией языка JavaScript, поддерживаемой в старых браузерах;
- оптимизировать JavaScript с удалением ненужного кода и минимизацией инструкций.

Webpack — невероятно мощный инструмент, с которым работают многие команды во многих компаниях. Как было сказано ранее в этой главе, здесь я не буду подробно рассказывать, как его применять. Надеюсь, вы самостоятельно изучите дополнительные возможности React и связанных инструментов сборки. Изучение их излишне усложнит книгу, а обучение не упростит. Процесс сборки, включенный в исходный код, можно понять, если уделить некоторое время ознакомлению с информацией о Webpack на странице webpack.js.org.

Вы начнете разработку программы Letters Social, создав компонент App и основной индексный файл, который будет служить точкой входа в приложение (где вызывается метод `render` из `React DOM`). Компонент App будет содержать некоторую логику для получения сообщений из API и станет рендерить несколько компонентов Post — компонент для сообщений вы создадите позднее. Репозиторий содержит также ряд компонентов, которые не нужно создавать самостоятельно. Вы будете использовать их сейчас и в следующих главах. В листинге 4.7 показан файл точки входа src/index.js.

Основной файл приложения содержит ссылки на некоторые стили, которые Webpack может импортировать, а также основной вызов метода `render React DOM`. Это основная позиция, с которой ваше React-приложение стартует. Когда сценарий будет запущен браузером, он отобразит основное приложение и React возьмет работу на себя. Без этого вызова приложение запущено не будет. Вы, вероятно, помните из предыдущих глав, что указывали вызов в нижней части основного файла приложения.

Здесь все иначе: приложение будет состоять из множества разных файлов, и Webpack будет знать, как их объединить (благодаря инструкциям `import/export`) и запускаться в браузере.

Листинг 4.7. Основной файл приложения (src/index.js)

```
import React, { Component } from 'react';
import { render } from 'react-dom';

import App from './app';

import './shared/crash';
import './shared/service-worker';
import './shared/vendor';
import './styles/styles.scss';

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

Импорт React и метод рендеринга из React DOM — этот файл будет основным вызовом метода render React DOM

Импорт некоторых файлов, связанных с отчетами об ошибках, регистрацией сервис-воркера и стилистикой (поддерживаемой настройкой репозитория)

Импорт экспорта по умолчанию из компонента App — вы создадите его в следующем листинге

Вызов метода render с основным приложением в целевом элементе (HTML-шаблон находится в файле src/index.ejs)

Теперь, когда у вас есть точка входа для приложения, создадим основной компонент App. Можете поместить этот файл в каталог `src`, то есть `src/app.js`. Вы набросаете базовый каркас для компонента App, а затем по мере продвижения заполните его. В этой главе ваша цель — запустить основное приложение и отобразить несколько сообщений. В следующей главе вы начнете добавлять функции и реализуете возможность создавать сообщения, а также добавлять локации в сообщения. По мере изучения различных тем вы будете вводить в приложение все больше функциональности, например тестирование, маршрутизацию и архитектуру приложений (с использованием Redux). В листинге 4.8 показаны основы компонента App.

Листинг 4.8. Создание компонента App (src/app.js)

```
import React, { Component } from 'react';
import PropTypes from 'prop-types';
import parseLinkHeader from 'parse-link-header';
import orderBy from 'lodash/orderBy';

import ErrorMessage from './components/error/Error';
import Loader from './components/Loader';
import * as API from './shared/http';
import Ad from './components/ad/Ad';
import Navbar from './components/nav/navbar';
import Welcome from './components/welcome/Welcome';

class App extends Component {
  constructor(props) {
```

Импортирование библиотек, необходимых компоненту App

Импортирование сообщения об ошибке и компонентов загрузчика

Импортирование существующих компонентов Ad, Welcome и Navbar

Импортирование модуля API Letters для использования при создании и извлечении сообщений

```
    super(props);
    this.state = {
      error: null,
      loading: false,
      posts: [],
      endpoint: '${process.env
.ENDPOINT}/posts?_page=1&_sort=date&_order=
   DESC&_embed=comments&_expand=user&_embed=likes'
    };
  }
  static propTypes = {
    children: PropTypes.node
  };
  render() {
    return (
      <div className="app">
        <Navbar />
        {this.state.loading ? (
          <div className="loading">
            <Loader />
          </div>
        ) : (
          <div className="home">
            <Welcome />
            <div>
              <button className="block">
                Load more posts
              </button>
            </div>
            <div>
              <Ad
                url="https://ifelse.io/book"
                imageUrl="/static/assets/ads/ria.png"
              />
              <Ad
                url="https://ifelse.io/book"
                imageUrl="/static/assets/ads/orly.jpg"
              />
            </div>
          </div>
        )}
      </div>
    );
  }
}

export default App;
```

Настройка начального состояния компонента — вы будете отслеживать сообщения и конечную точку, чтобы увидеть больше сообщений

При загрузке рендерится загрузчик, а не тело приложения

Здесь добавляются компоненты для отображения сообщений

Рендеринг компонентов Welcome и Ad

Экспорт компонента App

Имея все это, вы можете выполнить команду разработки (`npm run dev`), и ваше приложение должно загрузиться и быть доступным в браузере. Если вы этого еще не сделали, выполните команду `npm run db: seed` по крайней мере однократно, чтобы

сгенерировать образцы данных для базы данных. Запуск `npm run dev` даст следующие результаты:

- запустятся процесс сборки Webpack и сервер разработки;
- запустится API JSON-сервера, чтобы отвечать на сетевые запросы;
- будет создан сервер разработки (пригодится для рендеринга на стороне сервера в главе 12);
- произойдет горячая перезагрузка приложения при возникновении изменений (так что не нужно обновлять приложение при каждом сохранении файла);
- будут выведены сообщения об ошибках сборки (они станут отображаться в командной строке и браузере).

Когда приложение запущено в режиме разработки, у вас должен быть доступ к нему по адресу http://localhost:3000. Сервер API доступен по адресу http://localhost:3500, вы можете выполнять запросы к нему с помощью таких инструментов, как Postman (www.getpostman.com), или переходить к различным ресурсам с помощью браузера.

С учетом этих логистических вопросов вы должны добавить в компонент App возможность получать сообщения. Для этого нужно отправить сетевой запрос в API Letters Social с помощью API Fetch (в комплекте с включенным модулем API). Сейчас ваш компонент делает немногое. Вы не определили какие-либо методы жизненного цикла вне методов `constructor` и `render`, поэтому у компонента нет данных для работы. Вам следует получить данные из API, а затем обновить состояние компонента с ними. Вы также добавите границы ошибки, чтобы при обнаружении компонентом ошибки можно было отобразить сообщение об ошибке, а не размонтировать все приложение. В листинге 4.9 показано, как добавить методы класса в компонент App.

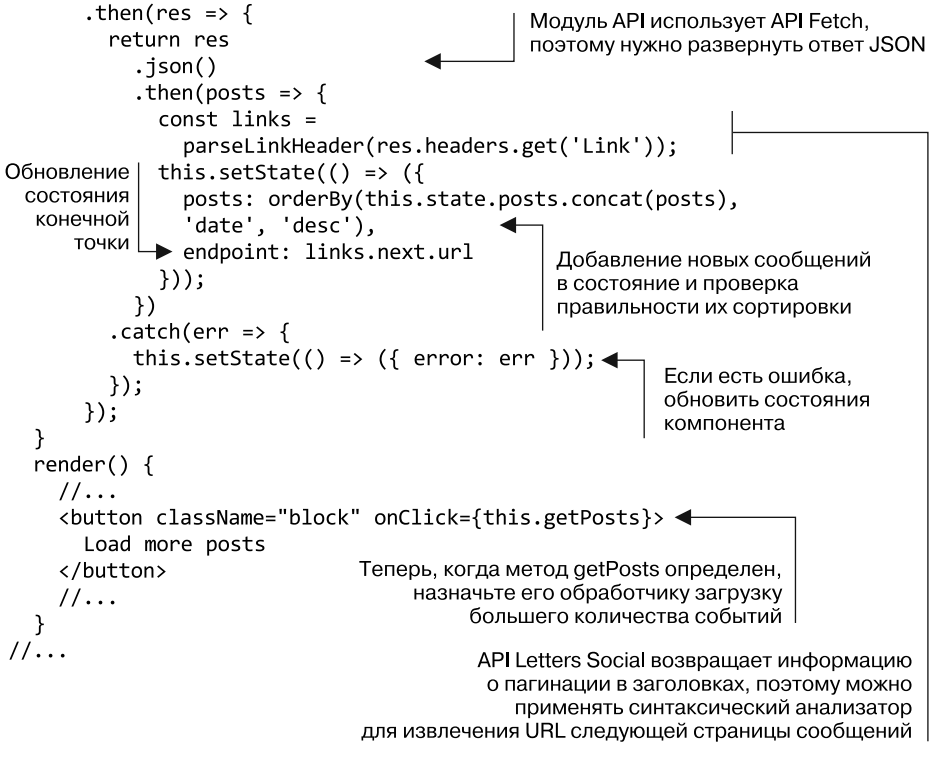
Листинг 4.9. Получение данных при монтировании компонента App

```
//...
  constructor(props) {
    //...
    this.getPosts = this.getPosts.bind(this);
  }

  componentDidMount() {
    this.getPosts();
  }

  componentDidCatch(err, info) {
    console.error(err);
    console.error(info);
    this.setState(() => ({
      error: err
    }));
  }
  getPosts() {
    API.fetchPosts(this.state.endpoint)
```

Связывание и использование метода класса для получения сообщений из API, когда компонент монтируется

Настройка границы ошибки для приложения, чтобы вы могли обрабатывать ошибки

Выбор сообщения с помощью включенного модуля API



Приложение должно получать сообщения, когда монтируется, и сохранять эти данные в состоянии локального компонента. Затем нужно создать компонент Post, в котором будут размещены данные сообщений. Вы создадите его из набора уже существующих компонентов, поставляемых с исходным кодом. Это в основном функциональные компоненты без состояния, и в дальнейшем вы будете выполнять разработку на их основе. Проанализируйте каталог `src/components/post`, чтобы ознакомиться с ними.

Ваши сообщения также будут загружать собственный контент, чтобы в следующих главах вы могли перемещать компонент Post и рендерить его. Компонент App выполняет запрос на получение сообщений, но все, что ему действительно требуется, — это идентификатор и дата публикации, тогда как сам компонент Post будет отвечать за загрузку остальной части содержимого. Другой способ сделать это заключается в том, чтобы компонент App стал ответственным за полное извлечение данных и просто передавал данные в сообщение. Один из положительных моментов при таком подходе заключается в том, что выполняется меньшее количество сетевых запросов. Вы делаете сообщение ответственным за получение дополнительных данных для иллюстрации, а также потому, что мы все еще фокусируемся на методах жизненного цикла обучения. Но чтобы стало яснее, я хотел бы рассмотреть и другой подход. В листинге 4.10 показан компонент Post. Создайте его в файле `src/components/post/Post.js`.

Листинг 4.10. Создание компонента Post (src/components/post/Post.js)

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

import * as API from '../../shared/http';
import Content from './Content';
import Image from './Image';
import Link from './Link';
import PostActionSection from './PostActionSection';
import Comments from '../comment/Comments';
import Loader from '../Loader';

export class Post extends Component {
  static propTypes = {
    post: PropTypes.shape({
      comments: PropTypes.array,
      content: PropTypes.string,
      date: PropTypes.number,
      id: PropTypes.string.isRequired,
      image: PropTypes.string,
      likes: PropTypes.array,
      location: PropTypes.object,
      user: PropTypes.object,
      userId: PropTypes.string
    })
  };
  constructor(props) {
    super(props);
    this.state = {
      post: null,
      comments: [],
      showComments: false,
      user: this.props.user
    };
    this.loadPost = this.loadPost.bind(this);
  }
  componentDidMount() {
    this.loadPost(this.props.id);
  }
  loadPost(id) {
    API.fetchPost(id)
      .then(res => res.json())
      .then(post => {
        this.setState(() => ({ post }));
      });
  }
  render() {
    if (!this.state.post) {
      return <Loader />;
    }
    return (
```

Импортирование модуля API, чтобы можно было получить сообщение

Импортирование составных компонентов для Post

Необходимы методы жизненного цикла, поэтому расширяется React.Component

Объявление propTypes

Определение конструктора, чтобы можно было установить состояние и связать методы класса

Задание начального состояния

Привязка метода класса

Загрузка сообщения при монтировании

Использование API, чтобы получить одно сообщение и обновить состояние

Если сообщение еще не загружено, отобразить компонент загрузчика

```
        <div className="post">
          <UserHeader date={this.state.post.date}
                      user={this.state.post.user} />
          <Content post={this.state.post} />
          <Image post={this.state.post} />
          <Link link={this.state.post.link} />
          <PostActionSection showComments={this.state.showComments}/>
          <Comments
            comments={this.state.comments}
            show={this.state.showComments}
            post={this.state.post}
            user={this.props.user}
          />
        </div>
      );
    }
  }

  export default Post;
```

Настройка компоновки (разметки) для компонента CommentBox

Последнее, что вам нужно сделать, — перебрать сообщения, чтобы они отображались. Помните, как можно отобразить динамический список компонентов, — сформировать массив (с помощью `Array.map` или другим способом) и задействовать его в выражении JSX. Не забывайте также, что React требует передачи ключевого свойства каждому перебираемому элементу, чтобы он знал, какие компоненты нужно обновить в динамическом списке. Это справедливо для любого массива компонентов, возвращаемого в методе `render`. В следующем листинге показано, как обновить метод `render` компонента App, чтобы перебирать сообщения.

Листинг 4.11. Итерация по компонентам Post (src/app.js)

```
//...
import Post from './components/post/Post';
//...
<Welcome />
```

Импортирование компонента Post

```
<div>
  {this.state.posts.length && (
    <div className="posts">
      {this.state.posts.map(({ id }) => (
        <Post id={id} key={id}
        user={this.props.user} />
      ))}
    </div>
  )}
  <button className="block" onClick={this.getPosts}>
    Load more posts
  </button>
</div>
<div>
  <Ad
    url="https://ifelse.io/book"
```

Итерации по извлеченным сообщениям и отображение компонента Post для каждого из них

Обязательное добавление ключевого свойства каждому перебираемому элементу

```
            imageUrl="/static/assets/ads/ria.png"
          />
          <Ad
            url="https://ifelse.io/book"
            imageUrl="/static/assets/ads/orly.jpg"
          />
        </div>
//...
```

При этом вы рендерите сообщения и начинаете с Letters Social (рис. 4.10). Разумеется, существует много возможностей для усовершенствования, но мы пошли таким путем.

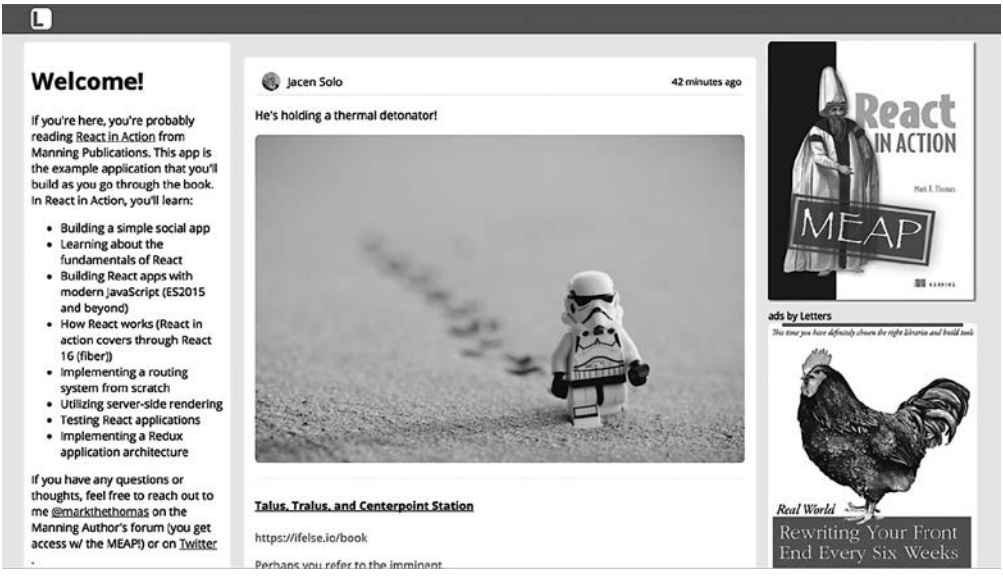


Рис. 4.10. Первый прогон Letters Social. Сообщения отображаются, и вы можете загрузить дополнительные. В следующей главе добавите возможность создавать сообщения с местоположением

В главе 5 мы рассмотрим добавление сообщений и локаций в сообщения. А также освоим использование ссылок — способ доступа к базовым элементам DOM из компонентов React.

Упражнение 4.3. Неперехваченные ошибки

Что происходит, если непойманная ошибка возникает в компоненте React? Есть ли способы обработки ошибки?

4.4. Резюме

Повторим то, что вы узнали из этой главы.

- Компоненты React создаются из классов JavaScript, которые наследуются от класса `React.Component` и имеют жизненный цикл, к которому вы можете подключиться. Это означает, что у них есть начало, середина и конец времени существования, которым управляет React. Поскольку они наследуются от абстрактного базового класса `React.Component`, то имеют доступ к специальным API React, чего нет у функциональных компонентов без состояния.
- React обеспечивает методы жизненного цикла, с помощью которых можно подключаться к различным частям жизни компонента. Это позволяет приложению действовать надлежащим образом в разных частях процесса React, управляющего пользовательским интерфейсом. Не все методы жизненного цикла следует применять, их нужно вводить только в случае необходимости. Во многих случаях все, что вам нужно, — это функциональный компонент без состояния.
- React предоставляет метод обработки ошибок, которые возникают в конструкторе, в процессе рендеринга или в методах жизненного цикла компонента — `componentDidCatch`. С его помощью задают *границы ошибок* в приложении. Это подобно конструкции `try...catch` в языке JavaScript. Если React обнаруживает ошибку, он размонтирует компонент, в котором она произошла, и его потомков из DOM, чтобы повысить стабильность рендеринга и предотвратить сбой всего приложения.
- Вы начали создавать Letters Social — проект, который мы будем использовать для изучения React в остальной части книги. Окончательная версия проекта доступна по адресу social.react.sh, а исходный код размещен по адресу github.com/react-in-action/letters-social.

В следующей главе вы начнете добавлять новые функциональные возможности в приложение Letters Social. Мы сосредоточимся на динамическом создании сообщений и внедрении в них локаций с помощью инструмента Mapbox.

5 Работа с формами в React

- Использование форм в React.
- Контролируемые и неконтролируемые компоненты формы в React.
- Проверка и очистка данных в React.

К этому моменту вы освоили некоторые приемы создания простых компонентов с помощью React: подключение к жизненному циклу, PropTypes и API компонента верхнего уровня. Вы понимаете основные принципы работы в ней и можете выполнять базовые действия, например обновлять локальное состояние компонента и передавать данные между компонентами с помощью свойств. А также познакомились со структурой компонентов, разобрались в относящейся к ним терминологии и методах жизненного цикла.

В этой главе вы расширите свои знания и начнете разработку демонстрационного приложения Letters Social: выполните компонент, который пользователи смогут применять для создания новых сообщений в приложении Letters Social. Сначала мы исследуем общую проблему и рассмотрим требования к данным. Затем поговорим о формах в React, и вы снабдите компонент функциональностью. В конце главы вы узнаете, как использовать формы в React-приложениях.

Получение исходного кода

Как и прежде, вы можете получить исходный код примеров из этой главы, перейдя в репозиторий GitHub по адресу github.com/react-in-action/letters-social. Если хотите начать работу здесь самостоятельно с нуля, возьмите исходный код примеров из главы 4 (если изучили ее и сами выполнили примеры) или обратитесь к ветви, относящейся к данной главе (chapter-5-6).

Помните, что каждая ветвь содержит итоговый код главы (например, chapter-5-6 содержит код, получаемый в конце глав 5 и 6). Вы можете выполнить в оболочке командной строки одну из следующих команд по своему выбору, чтобы получить код примеров из текущей главы. Если репозитория вообще нет, выполните такую команду:

```
git clone git@github.com:react-in-action/letters-social.git
```

Если у вас уже клонирован репозиторий, то следующую:

```
git checkout chapter-5-6
```

Возможно, вы перешли сюда из другой главы, поэтому стоит проверить, установлены ли у вас нужные зависимости, с помощью следующей команды:

```
npm install
```

5.1. Создание сообщений в Letters Social

В данный момент React-приложение Letters Social позволяет только читать сообщения, и этим его возможности ограничиваются. Социальная сеть «только для чтения» больше похожа на библиотеку, и это не совсем то, чего хотят ваши вымышленные инвесторы. Первая функция, которую нужно реализовать, — возможность создания сообщений. Пользователи смогут писать сообщения с помощью форм и выводить их в ленте новостей. Прежде чем начать работу, рассмотрим требования к данным и проанализируем проблему, чтобы вы поняли, что нужно сделать.

5.1.1. Требования к данным

Вы будете использовать определенные HTTP-библиотеки браузера для отправки данных на фиктивный сервер API. Вероятно, вы уже знаете, как они работают и как обращаться с RESTful и другими веб-API языка JavaScript, поэтому я не буду подробно рассказывать о них. Если у вас нет опыта работы с протоколом HTTP в браузере или с серверами, получите информацию из множества отличных источников, например из книги *JavaScript Application Design* Николаса Бевакквы (Nicolas G. Bevacqua) (Manning, 2015).

Работая с API, вы обычно должны отправлять данные, которые соответствуют контракту. Если база данных ожидает информацию о пользователе, может потребоваться включить такие сведения, как ваши имя, адрес электронной почты и, опционально, изображение профиля. Эти данные, как правило, должны иметь определенный формат, иначе сервер отклонит их. Первым делом следует выяснить формат данных, поддерживаемых сервером.

В листинге 5.1 показана базовая схема для сообщений в Letters Social. Мы задействуем простой класс JavaScript, поскольку именно его будет использовать сервер. При создании сообщения данные, которые вы отправляете на сервер, должны содержать большинство элементов, указанных в схеме. Обратите внимание на то, что сообщение может содержать множество полезных сведений, в том числе о местоположении (вы займетесь добавлением местоположения в главе 6). Сервер автоматически присвоит значения по умолчанию одним неуказанным свойствам и проигнорирует другие. Чего точно не нужно делать в браузере — создавать уникальный идентификатор, так как сервер делает это самостоятельно.

Листинг 5.1. Схема Post (db/models.js)

```
export class Post {
  constructor(config) {
    this.id = config.id || uuid();
    this.comments = config.comments || [];
    this.content = config.content || null;
    this.date = config.date || new Date().getTime();
    this.image = config.image || null;
    this.likes = config.likes || [];
    this.link = config.link || null;
    this.location = config.location || null;
    this.userId = config.userId;
  }
}
```

5.1.2. Обзор и иерархия компонентов

Теперь, когда вы немного знаете о поддерживаемых данных, можно подумать о том, как выразить их в формате компонента. Существует множество аналогичных социальных приложений, поэтому сложностей в работе возникнуть не должно. На рис. 5.1 показан конечный интерфейс, который вы разработаете. Изучите его — пусть он вдохновит вас.

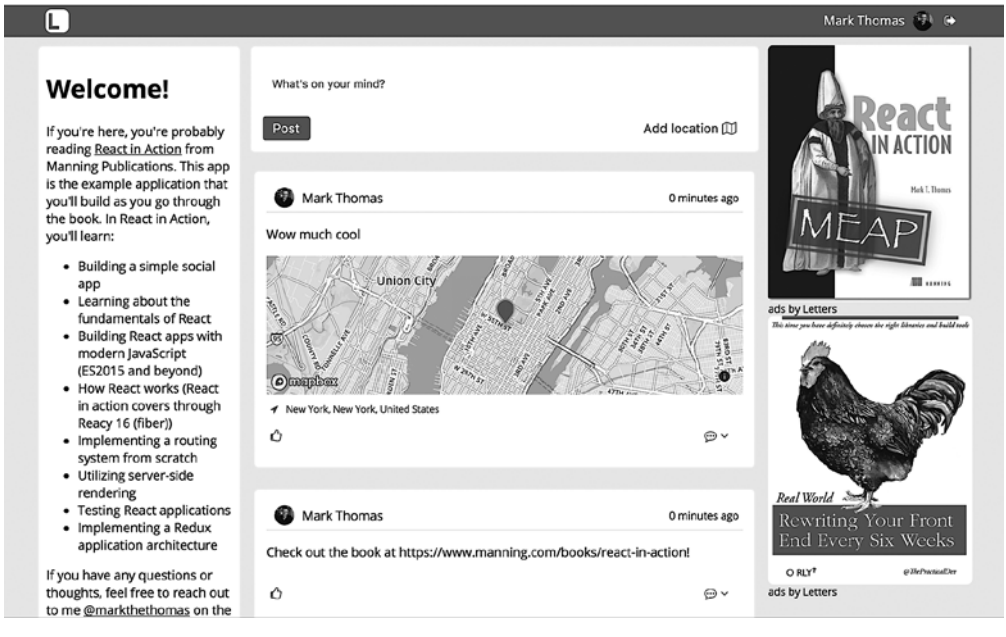


Рис. 5.1. Готовый интерфейс приложения Letters Social, которое будет создано. Можете ли вы разбить интерфейс на компоненты?

Я уже говорил об иерархии и отношениях компонентов и подчеркивал важность этих сведений для написания приложений с помощью React. Повторим эту информацию, прежде чем создавать компонент. Вот какие данные вы сейчас используете в приложении Letters:

- сообщения, доступные с помощью API. У некоторых из них есть изображения, у других — ссылки;
- пользовательские данные для каждого сообщения с какой-то информацией об аватаре;
- компонент App, который служит для сборки всего приложения;
- компонент Post, используемый при итерации по данным из API.

Вам следует добавить возможность создавать сообщения с поддержкой местоположения, а также текстовым содержимым. Нужно, чтобы пользователь смог выбирать свое местоположение, а затем указывать его в каждом сообщении в ленте новостей. Где должен находиться компонент `CreatePost`? Глядя на макеты и потребности пользователей, считаю, что имеет смысл разместить его как элемент того же уровня на итерирующихся сообщениях, полностью в основном компоненте App (рис. 5.2).

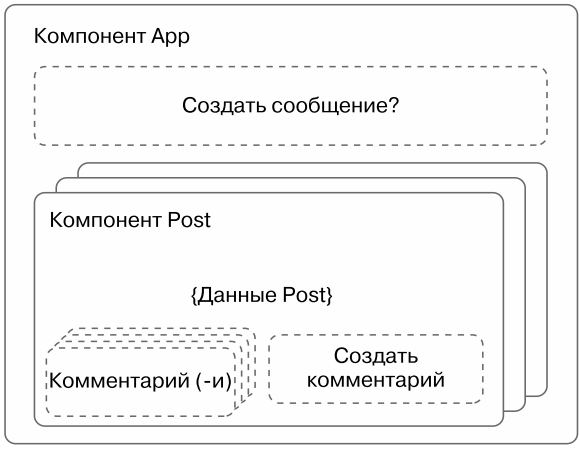


Рис. 5.2. Существующие и будущие компоненты. Вы создали компоненты Post и App, которые извлекают и перебирают данные. Компонент Create Post будет существовать вне компонентов, используемых для отображения сообщений

Рассмотрим, как разработать каркас компонента. Вы укажете только основы компонента, который станет рендерить базовый элемент, импортировать нужные инструменты, экспортировать класс `Component` и настраивать `PropTypes`, которые будут определены позже. В листинге 5.2 показано, как выполнить базовый каркас.

Листинг 5.2. Создание каркаса компонента (src/components/post/Create.js)

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

class CreatePost extends Component {
  static propTypes = {
  }
  constructor(props) {
    super(props);
  }
  render () {
    return (
      <div className="create-post">
        Create a post - coming (very) soon
      </div>
    );
  }
}

export default CreatePost;
```

Импорт объектов React и PropTypes, чтобы можно было их использовать

Создание компонента React

Объявление PropTypes в виде статического свойства класса

Настройка конструктора — вы будете применять его позже

Экспорт компонента, чтобы можно было использовать его в другом месте

5.2. Веб-формы в React

Оба компонента, которые вы разрабатываете в этой главе, задействуют формы. Веб-формы по-прежнему похожи на бумажные — они являются структурированным средством получения и записи ввода. Для записи информации на бумаге вы берете ручку или карандаш. В случае веб-форм применяете клавиатуру, мышь и файлы на компьютере для сбора информации. Существует несколько элементов веб-форм, с которыми вы, вероятно, знакомы, например `input`, `select` и `textarea`.

Большинство веб-приложений содержат те или иные формы. Я никогда не работал над приложением, которое было бы развернуто в рабочей среде и не содержало хотя бы одну веб-форму. Я также знаю, что формы имеют дурную репутацию, так как иногда с ними сложно работать. Возможно, именно по этой причине во многие фреймворки внедрен «магический» подход к веб-формам, который призван облегчить работу разработчика. React не использует такой подход, но может упростить работу с формами.

5.2.1. Начало работы с веб-формами

Не существует стандартного способа создать веб-формы с помощью клиентских фреймворков. В некоторых фреймворках и библиотеках вы можете настроить *модель* формы, которая обновляется, когда пользователь меняет значения, и имеет встроенные специальные методы для определения состояния формы. Другие фреймворки используют различные парадигмы и методы, когда дело доходит до форм. Их объединяет то, что они работают с веб-формами разными способами.

Что мы должны извлечь из разных подходов? Чем один лучше другого? Трудно сказать, лучше ли один другого, но иногда более простые в применении подходы

могут скрыть основные механизмы и логику от разработчика. Это не всегда плохо: иногда вам не стоит видеть внутреннюю работу фреймворка. Но нужно иметь достаточно информации о внутренней работе в рамках ментальной модели, чтобы создавать поддерживаемый код и исправлять ошибки при их возникновении. Вот где, на мой взгляд, проявляются лучшие качества библиотеки React. Не прибегая к излишней «магии», когда дело доходит до веб-форм, вы получаете отличное соотношение между необходимостью знать о формах слишком много или слишком мало.

К счастью, вы уже многое узнали о веб-формах в React. Не существует какого-либо специального набора интерфейсов API: формы — это просто еще одна демонстрация увиденного в React ранее — компонентов! Вы используете компоненты, состояние и свойства для создания форм. Поскольку мы основываемся на предыдущих главах, рассмотрим некоторые части ментальной модели React, прежде чем читать дальше.

- У компонентов есть два основных способа работы с данными — состояние и свойства.
- Будучи классами JavaScript, компоненты, в дополнение к подключению к жизненному циклу, могут иметь пользовательские методы класса, которые можно задействовать для реагирования на события и практически для всего прочего.
- Как и в случае обычных DOM-элементов, вы можете отслеживать события, такие как щелчки кнопкой мыши, изменения ввода и другие, в компонентах React.
- Родительские компоненты, такие как элементы формы, могут предоставлять методы обратного вызова в качестве свойств дочерних компонентов, что позволяет компонентам взаимодействовать друг с другом.

Вы будете использовать эти принципы React, когда начнете строить компонент для создания сообщений.

5.2.2. Элементы и события формы

Чтобы создать сообщение, вам необходимо убедиться, что оно сохраняется в базе данных, обновлен интерфейс сообщений и обновлен список сообщений для пользователя. Во-первых, вы обустроите элементы формы, которые будете выполнять, так же, как и в случае обычной HTML-формы. Разметки мало — вы получаете только ввод и не должны выводить что-то иное. В листинге 5.3 показано самое начало компонента — рендеринг ввода из элемента `textarea`.

Листинг 5.3. Добавление к компоненту CreatePost (src/components/post/Create.js)

```
//...
class CreatePost extends Component {
  render() {
    return (
      <div className="create-post">
        <textarea
          placeholder="What's on your mind?"
        />
```

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

Теперь, когда вы создали разметку каркаса основной формы, можете приступить к тонкой настройке. Как известно из предыдущих разделов, React позволяет взаимодействовать с событиями как в обычном JavaScript-коде в браузере. Библиотека позволяет отслеживать стандартные события, такие как щелчки кнопкой мыши, прокрутка и пр., и реагировать на них. Вы будете использовать эти события при работе с веб-формами.

ПРИМЕЧАНИЕ

Если вы уже работали над пользовательскими интерфейсами, то знаете, что между браузерами разных компаний существует много различий и несоответствий, особенно когда дело касается событий. В дополнение ко всему прочему, что вы получаете от React, библиотека позволяет абстрагироваться от этих различий в реализациях браузеров. Это неявное преимущество может здорово помочь. То, что не нужно беспокоиться о различиях между браузерами, позволяет сосредоточиться на других функциях приложения и, как правило, упрощает работу разработчика.

В результате взаимодействия пользователя с сайтом в браузере могут возникать различные события, в том числе перемещение мыши, ввод с клавиатуры, щелчки кнопкой мыши и многое другое. Нам интересны некоторые из этих типов событий в отношении приложения. Нужно слушать события с помощью двух главных обработчиков событий — `onChange` и `onClick`:

- `onChange` срабатывает при изменении элемента ввода. Можно получить новое значение элемента формы, используя значение `event.target.value`;
- `onClick` срабатывает при щелчке кнопкой мыши на элементе. Это событие прослушивают, чтобы определить, когда пользователь хочет отправить сообщение на сервер.

Затем вы назначите обработчики для этих событий. На данный момент можете реализовать некоторые консольные инструменты журналирования для этих функций, чтобы наблюдать за их запуском. Вы замените их реальными функциями позже. В листинге 5.4 показано, как можно настроить обработчики событий, привязывая их в конструкторе класса компонента и затем назначая их в компонентах.

Листинг 5.4. Добавление кода в компонент CreatePost (src/components/post/Create.js)

```
class CreatePost extends Component {
  constructor(props) {
    super(props);
    this.handleSubmit = this.handleSubmit.bind(this);
```

Методы класса Bind для обработки передаваемых данных и публикации изменений

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

  handlePostChange(e) {
    console.log('Handling an update to the post body!');
  }

  handleSubmit() {
    console.log('Handling submission!');
  }

  render() {
    return (
     <div className="create-post">
       <button onClick={this.handleSubmit}>Post</button>
       <textarea
         value={this.state.content}
         onChange={this.handlePostChange}
         placeholder="What's on your mind?"
       />
     </div>
   );
  }
}
```

Объявление метода обработки события отправки, React передаст событие обработчику

Объявление метода для класса, который будет использоваться при обновлении текста тела (событие onChange)

Передача обработчиков событий компонентам кнопки и текстовой области

Значение компонента будет считано из состояния компонента

Обработчики событий получают синтетическое событие в качестве аргумента, и мы получаем доступ к ряду свойств этого события. В табл. 5.1 перечислены некоторые свойства, доступные для синтетического события. Под *синтетическим* событием я подразумеваю, что React переводит событие браузера в код для работы в компонентах React.

Таблица 5.1. Свойства и методы, доступные для синтетического события в React

Свойство	Возвращаемый тип
bubbles	boolean
cancelable	boolean
currentTarget	DOMEventTarget
defaultPrevented	boolean
eventPhase	number
isTrusted	boolean
nativeEvent	DOMEvent
preventDefault()	
isDefaultPrevented()	boolean
stopPropagation()	

Продолжение ⇩

Таблица 5.1 *(продолжение)*

Свойство	Возвращаемый тип
isPropagationStopped()	boolean
target	DOMEventTarget
timeStamp	number
type	string

Прежде чем читать дальше, попробуйте добавить код `console.log (event)` в обработчик события изменения компонента `Post`. Если вы введете что-то в элемент `textarea` и откроете консоль разработчика в браузере, то увидите, что сообщения журналируются (пример приведен на рис. 5.3). Если вы проверяете эти объекты или пытаетесь получить доступ к некоторым свойствам, перечисленным в табл. 5.1, то должны получить информацию о событии. В нашем проекте будем работать со свойством `target`, которое вы получаете обратно. Помните: `event.target` — это просто ссылка на DOM-элемент, который отправил событие, как это было бы в обычном JavaScript-коде.

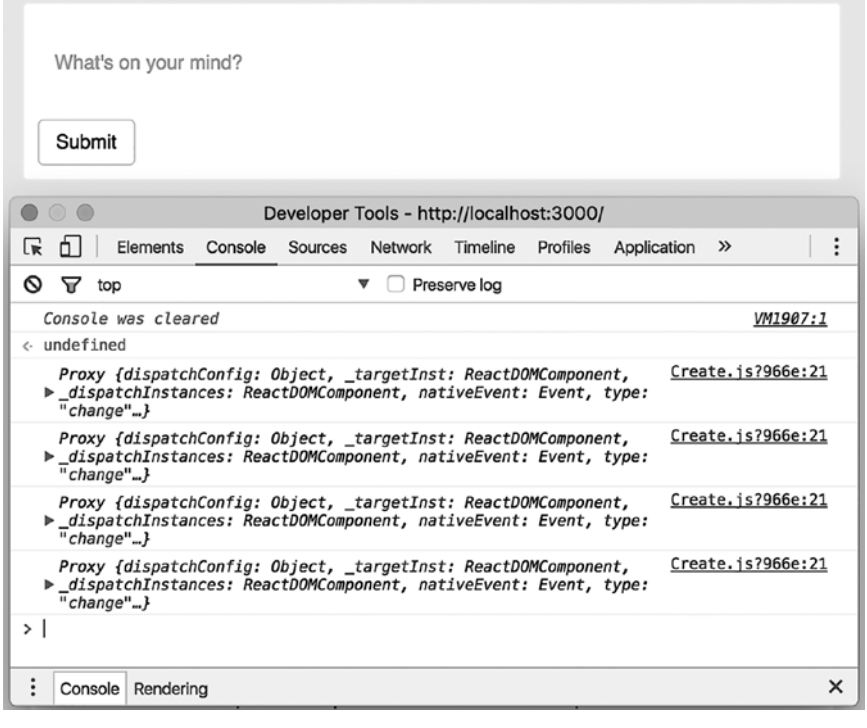


Рис. 5.3. React передает синтетическое событие обработчикам, которые вы настроили. Это нормализованное событие, то есть вы можете получить доступ к тем же свойствам и данным, которые доступны и для обычного события браузера

5.2.3. Обновление состояния в формах

Теперь можно прослушивать события и наблюдать, как компонент отслеживает обновления и события отправки данных, но пока ничего не делать с данными. Вам нужно что-то сделать с событиями, чтобы обновить состояние приложения. Это ключевой способ работы с формами в React: получение событий от обработчиков, а затем использование данных из этих событий для обновления состояния или свойств.

Состояние и свойства — два основных способа, с помощью которых React позволяет работать с данными. Если вы попытаетесь ввести что-либо в форму прямо сейчас, ничего не произойдет. Сначала это может показаться ошибкой, но React выполняет свою работу. Подумайте вот о чем: когда вы меняете вводимые данные, то изменяете DOM, а одна из основных задач React — убедиться, что DOM остается синхронизированной с версией DOM в памяти, которая создается из ваших компонентов.

Поскольку вы ничего не меняли в DOM в памяти (состояние не обновлялось), React не будет обновлять фактическую DOM с какими-либо изменениями. Это отличный пример прекрасной работы React. Если бы вы смогли обновить значения формы, то случайно поставили бы себя в сложную ситуацию, когда данные не синхронизированы и нужно вернуться к более старым способам работы (это то, что React улучшает в первую очередь).

Чтобы обновить состояние, нужно слушать событие, инициированное React при изменении ввода. Когда оно инициируется, вы извлекаете из него значение и задействуете его для обновления состояния компонента. Так можно контролировать каждый шаг процесса обновления.

Посмотрим, как реализовать это на практике. В листинге 5.5 показано, как настроить обработчики событий для прослушивания и обновления состояния компонентов, когда пользователь меняет вводимые данные. Позже вы станете использовать ссылку `event.target`, с которой работали ранее, и получите доступ к свойству `value`, чтобы обновить состояние с помощью значения из элемента `textarea`.

Листинг 5.5. Обновление состояния компонента с помощью вводов (src/components/post/Create.js)

```
class CreatePost extends Component {
  constructor(props) {
    super(props);

    // Установка состояния
    this.state = {
      content: '',
    };

    // Установка обработчиков событий
    this.handleSubmit = this.handleSubmit.bind(this);
    this.handlePostChange = this.handlePostChange.bind(this);
  }

  handlePostChange(event) {
    const content = event.target.value;
```

Захват значения элемента textarea из свойства value элемента DOM (то, с чем вы хотите обновить состояние)

```
      this.setState(() => {
        return {
          content,
        };
      });
  }

  handleSubmit() {
    console.log(this.state);
  }

  render() {
    return (
      <div className="create-post">
        <button onClick={this.handleSubmit}>Post</button>
          <textarea
            value={this.state.content}
            onChange={this.handlePostChange}
            placeholder="What's on your mind?"
          />
        </div>
      );
  }
}
```

Использование этого значения для установки состояния и обновления его с новым значением

Чтобы просмотреть обновленное состояние, нажмите кнопку отправки формы и проверьте консоль разработчика

Элементу присвоено новое значение для элемента textarea

5.2.4. Контролируемые и неконтролируемые компоненты

Такой подход к обновлению состояния компонента в формах позволяет строго контролировать, как происходят обновления, используя события и обработчики событий для обновления состояния. Это, вероятно, наиболее распространенный способ обработки форм в React. Компоненты, разработанные с учетом этого процесса, обычно называют *контролируемыми* компонентами. Это потому, что мы жестко контролируем компонент и то, как изменяется состояние. Есть и еще один способ разработки компонентов с формами — *неконтролируемые* компоненты. На рис. 5.4 показано, как работают контролируемые и неконтролируемые компоненты, и приведены некоторые их различия.

Неконтролируемый компонент вместо использования свойства `value` для установки данных сохраняет собственное внутреннее состояние. Вы, как и прежде, можете слушать обновления ввода с помощью обработчика событий, но больше не способны управлять состоянием ввода. В листинге 5.6 показан пример применения неконтролируемого компонента. И хотя в книге мы будем работать с контролируемыми компонентами, важно хотя бы знать, как выглядит этот шаблон на практике.

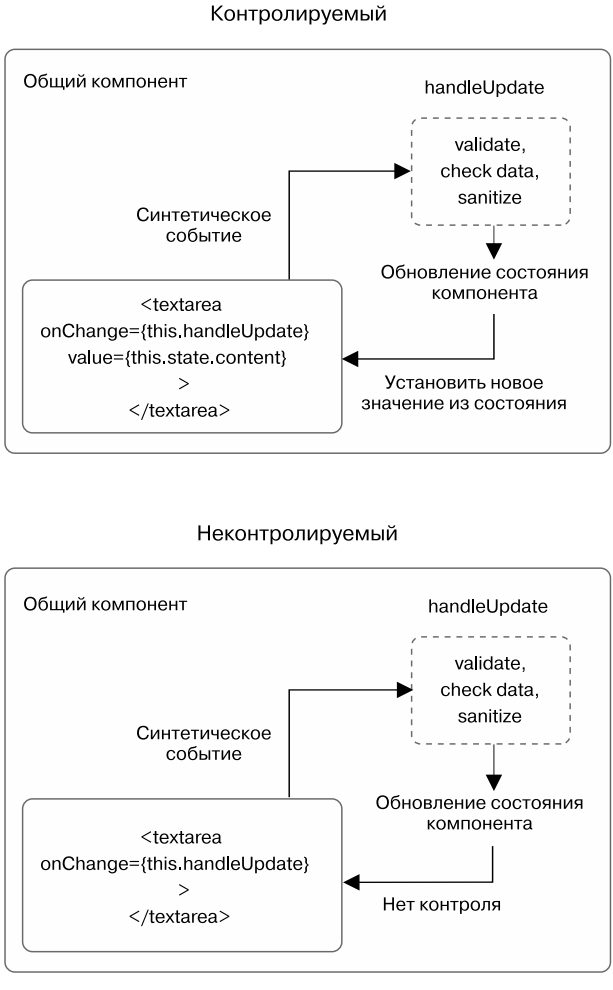


Рис. 5.4. Контролируемые компоненты прослушивают события, инициируемые элементом DOM, работают с данными, а затем обновляют состояние компонента и устанавливают значение элемента. Данные сохраняются в области компонента, и создается область унифицированного состояния. Неконтролируемые компоненты сохраняют собственное внутреннее состояние и формируют ситуацию, когда в компоненте существует микрокосм, прерывая доступ к этому состоянию и его контроль

Листинг 5.6. Использование неконтролируемых компонентов (src/components/post/Create.js)

```
class CreatePost extends Component {
  constructor(props) {
    super(props);

    this.state = {
```

```
      content: '',
    };

    this.handleSubmit = this.handleSubmit.bind(this);
    this.handlePostChange = this.handlePostChange.bind(this);
  }

  handlePostChange(event) {
    const content = event.target.value;
    this.setState(() => {
      return {
        content,
      };
    });
  }

  handleSubmit() {
    console.log(this.state);
  }

  render() {
    return (
      <div className="create-post">
        <button onClick={this.handleSubmit}>Post</button>
          <textarea
            onChange={this.handlePostChange}
            placeholder="What's on your mind?"
          />
        </div>
      );
  }
}
```

Обработчики такие же, как раньше, но эффект, который имеет изменяемое состояние, будет иным

Обработчики такие же, как раньше, но эффект, который имеет изменяемое состояние, будет иным

Как уже отмечалось, теперь нет элемента value, который будет прослушивать состояние компонента

5.2.5. Подтверждение и очистка формы

Один важный момент, касающийся использования форм для записи и хранения пользовательского ввода, — нужно позволить пользователям узнать, когда они нарушают какие-либо настроенные вами правила проверки и когда делают что-то, чтобы предоставить данные, не удовлетворяющие приложению. Необходимо, чтобы серверные приложения, которые получают данные из вашего клиентского приложения, имели строгие процедуры проверки и обработки данных, — вы не можете полагаться на браузер, чтобы выполнить всю работу в этой области. И даже если у вас корректно настроены очистка данных и проверка на месте на сервере, все равно необходимо обеспечить применение хороших практик в области данных на стороне пользователя и настоять на этом, чтобы помочь пользователям добавить еще один уровень защиты от недобросовестных участников процесса и обеспечить целостность данных. Если этого не сделать, потенциально у вас будут недовольные пользователи, дыры в безопасности и бессмысленные данные — все то, чего вы избегаете.

Как мы видели до сих пор, использование форм для обновления состояния компонента включает в себя состояние, свойства и методы компонента, как и многое другое в React. Чтобы добавить вашему компоненту возможность выполнения проверки и очистки, нужно подключиться к процессу обновления. С этой целью вы напишете общие функции проверки и очистки, которые можно применять везде, где поддерживается JavaScript, и, возможно, в большинстве других интерфейсных фреймворков.

Упражнение 5.1. Размышление о событиях и формах React

Задумайтесь на минутку о том, что вы узнали о событиях и формах в React. Отличаются ли события в React от событий, с которыми вы имели дело в браузере? И если это так, то в чем заключается различие?

К счастью, создаваемый компонент `CreatePost` не требует серьезной проверки. Вам нужно только проверить максимальную длину и еще кое-что, чтобы компонент не отправлял пустые сообщения на сервер API. Мы используем простую настройку сервера для обучения и локальной разработки, поэтому он будет принимать большинство полезных нагрузок без значительной проверки. Разработка серверных приложений — еще одна область, выходящая за рамки книги, поэтому я сосредоточусь только на проверке и очистке в браузере.

При настройке проверки форм и ввода в приложениях вы должны задать себе несколько вопросов.

- В чем состоят требования к данным для приложения?
- Исходя из ограничений форм, как вы можете помочь пользователям предоставлять значимые данные?
- Существуют ли способы устранения несоответствий в данных, предоставляемых пользователями?

Вначале нужно выяснить, какие требования к данным предъявляют бизнес- или прикладная стороны (если таковые существуют). Вы должны начать с этого, потому что это знание поможет вам определить основные рекомендации по исправлению данных. Поскольку мы уже установили, что сервер будет охотно принимать большинство данных, и назвали основные типы данных для сообщения, можем перейти к следующему вопросу.

Как с учетом определенных ограничений вы можете лучше всего помочь своим пользователям предоставлять значимые данные и успешно использовать ваше приложение? Обычно это касается проверки данных для таких свойств, как размер, тип символа, возможно, тип загружаемых файлов, и многих других. Сейчас компонент CreatePost довольно качественный, и, помимо длины, нужно проверять не так уж много. Затем вы проверите минимальную и максимальную длину и разрешите пользователю отправить свое сообщение, если оно допустимо. В листинге 5.7 показано, как настроить базовую проверку компонента.

Листинг 5.7. Добавление базовой проверки (src/components/post/Create.js)

```
//...
class CreatePost extends Component {
  constructor(props) {
    super(props);

    this.state = {
      content: '',
      valid: false,
    };

    this.handleSubmit = this.handleSubmit.bind(this);
    this.handlePostChange = this.handlePostChange.bind(this);
  }

  handlePostChange(event) {
    const content = event.target.value;
    this.setState(() => {
      return {
        content,
        valid: content.length <= 280
      };
    });
  }

  handleSubmit() {
    if (!this.state.valid) {
      return;
    }
    const newPost = {
      content: this.state.content,
    };
    console.log(this.state);
  }

  render() {
    return (
      <div className="create-post">
        <button onClick={this.handleSubmit}>Post</button>
          <textarea
            value={this.state.content}
            onChange={this.handlePostChange}
            placeholder="What's on your mind?"
          />
      </div>
    );
  }
}
```

Создание простого свойства valid в локальном состоянии компонента

Определение допустимости сообщения путем установки максимальной длины. Здесь 280 знаков — демонстрация примера, но иногда пользователи хотят, чтобы сообщения были длинными

Создание нового объекта публикации

Мы проработали ответы на первые два вопроса (ограничения и проверка данных). Теперь можем подойти к заключительному этапу — устранению несогласованности данных и их базовой очистке. Принимая во внимание, что при проверке

у пользователя запрашивают *определенные* сведения, очистка гарантирует, что данные, которые вы получите, безопасны, имеют правильный формат и могут быть сохранены. Информационная безопасность — огромная и очень важная сфера. Эта книга не может раскрыть все нюансы надлежащей обработки данных для обеспечения безопасности, но мы можем решить одну маленькую проблему для Letters — исключить нецензурную лексику.

Вы примените модуль JavaScript, называемый bad-words, доступный в менеджере npm (реестр модулей и служб JavaScript, см. www.npmjs.com/about). Он должен быть заранее установлен в вашем проекте. bad-words принимает строку и заменяет звездочками любые слова, найденные в черном списке (если хотите, создайте собственный список и замените им используемый по умолчанию). Пример, проиллюстрированный в листинге 5.8, в основном придуман, но, рассмотрев его, вы можете запретить людям публиковать потенциально оскорбительное содержимое в общедоступном приложении (social.react.sh). Помните, что это придуманный пример (мы не предлагаем и не поддерживаем любую цензуру).

Листинг 5.8. Добавление базовой очистки содержимого (src/components/post/Create.js)

```
import PropTypes from 'prop-types';
import React from 'react';

import Filter from 'bad-words';

const filter = new Filter();

class CreatePost extends Component {
  //...
  handlePostChange(event) {
    const content = filter.clean(event.target.value);
    this.setState(() => {
      return {
        content,
        valid: content.length <= 280
      };
    });
  }
//...
  }
export default CreatePost;
```

Импорт объекта по умолчанию из модуля bad-words

Применение конструктора для создания нового экземпляра фильтра

Передача значения формы методу .clean() фильтра и использование возвращаемого значения для установки состояния

5.3. Создание новых сообщений

Теперь, когда базовая проверка и очистка сообщений могут быть выполнены, необходимо создать их путем отправки на сервер. Мы собираемся немного усложнить эту задачу, поэтому вкратце изучим каждый шаг, а затем рассмотрим процесс в целом.

Чтобы отправить сообщение в API, нужно будет наряду с тем, что уже делает компонент CreatePost, отследить состояние, выполнить базовые проверку и очистку содержимого.

Затем, чтобы отправить данные API, требуется сделать следующее.

1. Захватить пользовательский ввод для использования в качестве сообщения, обновить состояние и выполнить логику проверки данных, которую вы реализовали ранее.
2. Вызвать функцию обработчика событий, переданную из родительского компонента (в нашем случае основного компонента приложения) в качестве свойства, и передать ему сообщения.
3. Сбросить состояние компонента `CreatePost`.
4. В родительском компоненте применять данные, переданные из дочернего компонента `CreatePost`, для передачи HTTP-запроса POST на сервер.
5. Обновить состояние локального компонента с новым сообщением, которое вы получите от сервера.

Чтобы лучше разобраться в том, что предстоит делать, рассмотрите рис. 5.5.

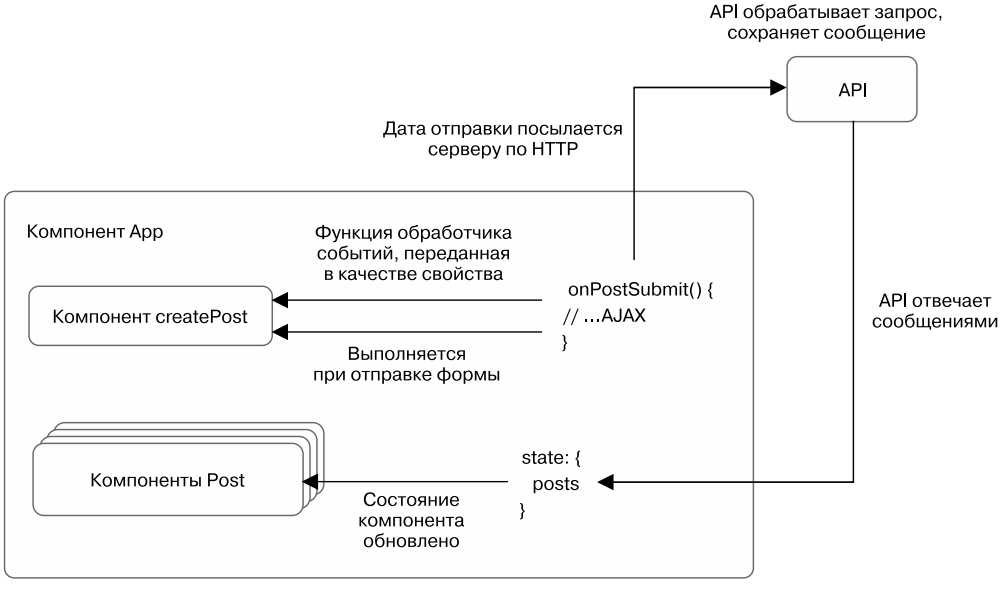


Рис. 5.5. Обзор компонента CreatePost. Компонент CreatePost получает функцию как свойство, использует свое внутреннее состояние как ввод для этой функции и вызывает ее, когда пользователь нажимает кнопку отправки данных. Эта функция, переданная из родительского компонента приложения, отправляет данные в API, обновляет локальные сообщения и инициирует обновление сообщений из API

Вы начнете с добавления функции, которая займется отправкой сообщений в родительском компоненте (`App.js`). У нее есть несколько составляющих, которые вы добавите по очереди, и мы изучим их все. В листинге 5.9 показано, как добавить функцию отправки сообщений в основной компонент `App`.

Листинг 5.9. Обработка передачи публикаций (src/app.js)

```
import * as API from './shared/http';

//...

export default class App extends Component {
  //...
  createNewPost(post) {
    this.setState(prevState => {
      return {
        posts: orderBy(prevState.posts.concat(newPost), 'date', 'desc')
      };
    });

  }
  //...
}
```

Импорт модуля API Letters

Конкатенация нового сообщения и проверка, что сообщения отсортированы

Вы настроили функцию обработчика post-creation в родительском компоненте, но в данный момент ничего не происходит, потому что ее никто не вызывает. Для работы следует передать ее дочернему компоненту (`CreatePost`, над которым вы работали). Помните, что можно передавать данные от предка потомку в качестве свойств? Вы также можете передавать функции. Это важно, так как компоненты способны сотрудничать и работать вместе. Хотя компоненты могут взаимодействовать, они не настолько связаны, чтобы вы не смогли их перемещать. К тому же легко перенести компонент `CreatePost` в другую часть приложения и передать те же данные другому обработчику. В листинге 5.10 показан пример передачи обратных вызовов в качестве свойств.

Упражнение 5.2 Контролируемые и неконтролируемые компоненты

Каковы различия между контролируемыми и неконтролируемыми компонентами в React? Как определить, считается компонент контролируемым или неконтролируемым?

Листинг 5.10. Передача обратных вызовов в качестве свойств (src/app.js)

```
import CreatePost from './post/Create';

export default class App extends Component {
  //...
  render() {
    return (
      //...
      <CreatePost onSubmit={this.createNewPost} />
      //...
    )
  }
  //...
}
```

Импорт компонента для использования

Передача функции handlePostSubmit с помощью свойств

На данном этапе вы настроили основы обработчика событий в родительском компоненте и передаете его в дочерний. Это помогает разобраться в проблемах — компонент `CreatePost` несет ответственность только за сбор некоторых данных публикаций и дальнейшую отправку их родительскому компоненту, чтобы тот передавал их в API. Об этом и многом другом рассказывается в главе 6.

5.4. Резюме

Вот основные моменты, которые вы узнали из этой главы.

- Формы обрабатываются в React во многом так же, как и любой другой компонент, — с применением событий и обработчиков событий для передачи и отправки данных.
- В React нет никаких магических способов работы с формами. Формы — это просто компоненты.
- Работу по проверке и очистке форм нужно проводить в рамках одной и той же ментальной модели событий React, обновлений компонентов, реиндексации, состояния и свойств и т. д.
- Вы можете передавать функции в качестве свойств между компонентами — это мощный и полезный шаблон проектирования, который предотвращает возникновение связанных компонентов, но способствует связи между компонентами.
- Проверка данных и очистка не являются магическими — React позволяет использовать для работы с данными обычный код JavaScript и библиотеки.

В следующей главе вы, опираясь на созданный проект, начнете интегрировать стороннюю библиотеку с React для добавления карт в свое приложение.

6 Интеграция сторонних библиотек с React

- Отправка данных формы в формате JSON удаленному API.
- Построение некоторых новых видов компонентов, в том числе для выбора географического местоположения, обработки ввода и отображения карты.
- Интеграция React-приложения с Mapbox для поиска местоположения и отображения карт.

В главе 5 мы начали изучать формы и их работу в React. Вы добавили обработчики событий для обновления состояния компонента в компоненте CreatePost. В данной главе вы, опираясь на выполненную работу, реализуете возможность добавления новых сообщений. Вы начнете теснее взаимодействовать с JSON API, который предоставил публикации для рендеринга в последней главе.

Зачастую вы будете создавать React-приложения с библиотеками, не относящимися к React, которые также работают с DOM. К ним могут относиться такие элементы, как jQuery, плагины jQuery или даже другие интерфейсные фреймворки. Мы видели, что React управляет DOM автоматически, упрощая работу с пользовательскими интерфейсами. Тем не менее возникают ситуации, когда нужно непосредственно взаимодействовать с DOM, что часто происходит при использовании сторонних библиотек. В этой главе мы рассмотрим некоторые способы достижения данного результата с помощью React при добавлении карт Mapbox в сообщения в приложении Letters Social.

Получение исходного кода

Как и прежде, вы можете найти исходный код примеров из этой главы, перейдя в репозиторий GitHub по адресу github.com/react-in-action/letters-social. Если хотите начать работу самостоятельно с нуля, возьмите исходный код примеров из главы 4 (если изучили ее и сами выполнили примеры) или обратитесь к ветви, относящейся к данной главе (chapter-5-6).

Помните, что каждая ветвь содержит итоговый код главы (например, chapter-5-6 содержит код, получаемый в конце глав 5 и 6). Вы можете выполнить в оболочке командной

строки одну из следующих команд по своему выбору, чтобы получить код примеров из текущей главы. Если репозитория вообще нет, выполните такую команду:

```
git clone git@github.com:react-in-action/letters-social.git
```

Если у вас уже клонирован репозиторий, то следующую:

```
git checkout chapter-5-6
```

Возможно, вы перешли сюда из другой главы, поэтому стоит проверить, установлены ли у вас все нужные зависимости, с помощью следующей команды:

```
npm install
```

6.1. Отправка сообщений в API Letters Social

В главе 2 вы создали компонент поля комментариев, который позволил добавлять комментарии. Он хранился локально, только в памяти, — в тот момент, когда страница обновляется, любые добавленные комментарии очищаются, потому что их жизнь и смерть зависят от ее состояния в данный момент. Вы могли бы использовать локальное или сеансовое хранилище или другую браузерную технологию хранения (например, cookie-файлы, IndexedDB, WebSQL и т. д.) — все они будут сохранять данные локально.

Вы отправите данные публикации в формате JSON на свой сервер API, как показано в листинге 6.1. Он будет поддерживать сохранение сообщения и отвечать новыми данными. Когда вы клонировали репозиторий, в папке `shared/http` уже были созданы функции, которые можно применять для проекта Letters Social. Вы используете библиотеку `isomorphic-fetch` для сетевых запросов. Она следует за API Fetch браузера, но имеет преимущество: может работать и на сервере.

Листинг 6.1. Отправка сообщений на сервер (src/components/app.js)

```
export default class App extends Component {
//...
createNewPost(post) {
  return API.createPost(post)
    .then(res => res.json())
    .then(newPost => {
      this.setState(prevState => {
        return {
          posts: orderBy(prevState.posts.concat(newPost),
            'date', 'desc')
        };
      });
    })
    .catch(err => {
      this.setState(() => ({ error: err }));
    });
}
```

Применение API Letters для создания сообщения

Получение ответа в формате JSON

Обновление состояния с использованием нового сообщения

Проверка того, что сообщения отсортированы методом OrderBy Lodash

Установка состояния ошибки, если она есть

При этом вам остается сделать только последнее: вызвать метод создания сообщений внутри дочернего компонента. Вы уже передали его, поэтому нужно просто проверить, что событие щелчка инициирует вызов родительского метода и данные сообщения передаются вместе. В листинге 6.2 показано, как вызвать метод, переданный в качестве свойства внутри дочернего компонента.

Листинг 6.2. Вызов функций, передаваемых через свойства

```
class CreatePost extends Component {

  // ...

  fetchPosts() {/* создана в главе 4 */}

  handleSubmit(event) {
    event.preventDefault();
    if (!this.state.valid) {
      return;
    }
    if (this.props.onSubmit) {
      const newPost = {
        date: Date.now(),
        // Присвоение публикации временного ключа; API создаст один
        id: Date.now(),
        content: this.state.content,
      };

      this.props.onSubmit(newPost);
      this.setState({
        content: '',
        valid: null,
      });
    }
  }
  // ...
}
```

Отмена события по умолчанию и создание объекта для отправки родительскому компоненту

Проверка того, что есть функция обратного вызова, с которой можно работать

Обратный вызов onSubmit через свойства из родительского компонента с передачей нового сообщения

Сброс состояния до первоначальной формы, чтобы отобразить для пользователя визуальную подсказку о том, что сообщение отправлено

Теперь если вы запустите приложение в режиме разработки, выполнив команду `npm run dev`, то сможете добавлять сообщения! Они должны появиться в ленте сразу, а если вы обновите свою страницу, то увидите и добавленную публикацию. Но не найдете изображения профиля и ссылок предварительного просмотра, как в других приложениях. Перечисленные функции добавим в последующих главах.

6.2. Расширение компонента с помощью карт

Теперь, когда вы обеспечили возможность создавать сообщения в приложении и отправлять их на сервер, можете немного расширить его функционал. Вымышленные инвесторы Letters Social пользовались сайтами Facebook и Twitter и заметили, что те позволяют добавлять в сообщения сведения о местоположении. Они хотят, чтобы приложение Letters Social тоже имело эту возможность, поэтому вы добавите

функцию выбора и отображения местоположения при выборе сообщения. А также повторно используете компонент отображения карты, чтобы сообщения в ленте новостей пользователя отражали географическое местоположение. На рис. 6.1 показано, что должно получиться в итоге.

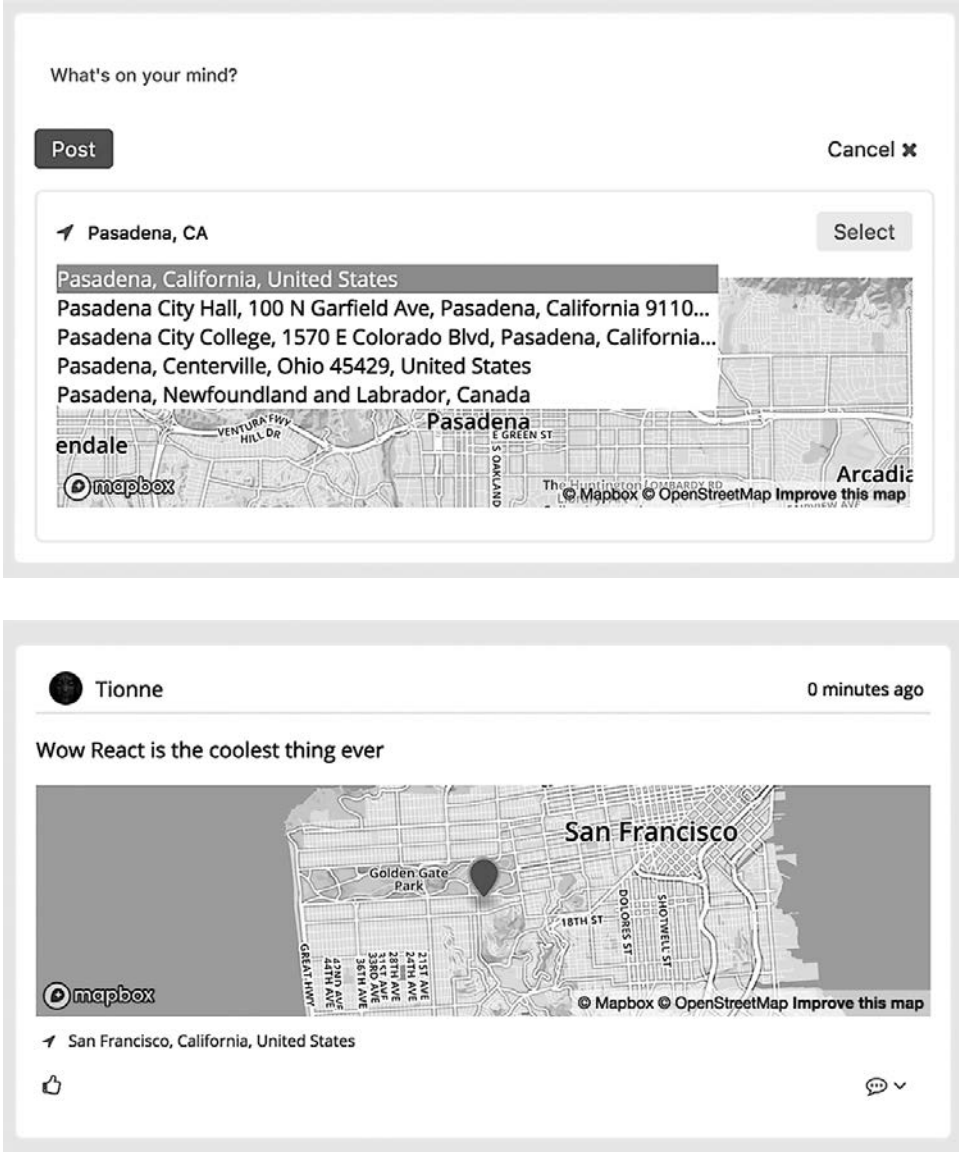


Рис. 6.1. Будущий функционал приложения Letters Social. Вы расширите его так, чтобы пользователи могли добавлять в свои публикации местоположение. Когда закончите, появится возможность поиска и выбора местоположения при создании сообщений

На рис. 6.1 видно, что для разработки карт используется инструмент Mapbox. Mapbox — это платформа картографирования и геосервисов, которая обеспечивает невероятное разнообразие услуг, связанных с картой и местоположением. Вы можете настраивать карты с данными, создавать разные стили карт и наложений, выполнять географический поиск, добавлять навигацию и многое другое. Я не могу описать все, что делает Mapbox, если хотите узнать больше, перейдите на сайт www.mapbox.com.

6.2.1. Разработка компонента DisplayMap с использованием ссылок

Вам понадобится способ отображения местоположения пользователя как при выборе местоположения для нового сообщения, так и при публикации сообщения в новостной ленте. Мы рассмотрим, как написать компонент, который будет служить обеим целям, чтобы можно было повторно задействовать свой код. Вам не всегда удастся это сделать, потому что для каждого места, где требуется карта, могут существовать разные требования. Но для данного случая совместное использование одного и того же компонента подойдет и сэкономит ваше время. Начните с создания файла с именем `src/components/map/DisplayMap.js`. Вы поместите в него оба компонента, связанных с картой.

Откуда появилась библиотека Mapbox? В большинстве случаев задействовали библиотеки, которые установили с помощью менеджера npm. Вы будете работать с модулем Mapbox в следующем подразделе, а для создания карт примените другую библиотеку. Если вы посмотрите HTML-код (`src/index.ejs`), то увидите ссылку на библиотеку Mapbox JS (`mapbox.js`):

```
...
<script src="https://api.mapbox.com/mapbox.js/v3.1.1/mapbox.js"></script>
...
```

Благодаря ей приложение React сможет работать с SDK Mapbox JS. Обратите внимание на то, что для SDK Mapbox JS требуется метка Mapbox. Я включил публичную метку в исходный код приложения Letters Social, поэтому вам не потребуется учетная запись Mapbox. Если у вас есть учетная запись или вы хотите ее создать для настройки под пользователя, добавьте свою метку, изменив значения в разделе конфигурации исходного кода приложения.

Часто возникают ситуации, когда вы работаете над проектом или функционалом, который требует интеграции React с библиотекой, не входящей в React. Возможно, это что-то вроде Mapbox (как в этой главе) или другая сторонняя библиотека, написанная без учета React. Учитывая, как React DOM управляет DOM, можете спросить: на что вы вообще способны влиять? Хорошая новость заключается в том, что в React есть несколько интересных обходных путей, которые позволяют работать с библиотеками такого типа.

Здесь в игру вступают ссылки. Я вскользь упоминал о ссылках в прошлых главах, а сейчас они будут особенно полезны. *Ссылка* — это способ React предоставить вам

доступ к базовому узлу DOM. Ссылки удобны, но не стоит злоупотреблять ими. Мы по-прежнему хотим использовать состояние и свойства в качестве основных средств для интерактивного взаимодействия наших приложений и работы с данными. Но в некоторых ситуациях ссылки полезны, в том числе:

- чтобы управлять фокусом и императивно взаимодействовать с элементами мультимедиа, такими как `<video>`;
- чтобы императивно вызывать анимацию;
- чтобы взаимодействовать со сторонними библиотеками, использующими DOM вне React (это наш случай).

Как задействовать ссылки в React? В предыдущих версиях вы добавили бы в элементы React строчный атрибут (`<div ref = "myref"> </ div>`), но новый подход заключается в применении встроенного обратного вызова, например:

```
<div ref={ref => { this.MyNode = ref; } }></div>
```

Когда вы обращаетесь к базовому DOM-элементу, вы можете сослаться на него из своего класса. Возможно взаимодействовать с ним в функции обратного вызова `ref`, но скорее вы захотите сохранить ссылку в своем классе компонента, чтобы она была доступна в другом месте.

Я должен отметить следующее: нельзя использовать ссылки в React для функционального компонента без состояния извне, потому что у него нет экземпляра поддержки. Например, это не сработает:

```
<ACoolFunctionalComponent ref={ref => { this.ref = ref; } } />
```

Но если компонент является классом, вы получаете ссылку на него, потому что он имеет экземпляр поддержки. Мможно также передавать ссылки в качестве свойств компонентам, которые их потребляют. В большинстве случаев вы станете использовать ссылки только тогда, когда нужен прямой доступ к узлу DOM, поэтому данный способ, вероятно, будет востребован нечасто, если только вы не собираете библиотеку, которая нуждается в ссылках для работы.

Вы собираетесь задействовать ссылки для взаимодействия с SDK Mapbox JavaScript. Библиотека Mapbox поддерживает создание карты без вашего участия и настраивает множество элементов, таких как обработчики событий, элементы управления пользовательским интерфейсом и т. д., на карте. Для API карты требуется применить ссылку на DOM-элемент или идентификатор для поиска DOM. Воспользуйтесь ссылкой. В листинге 6.3 показан каркас компонента `DisplayMap`.

Листинг 6.3. Добавление ссылок в компонент карты (src/components/map/DisplayMap.js)

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

export default class DisplayMap extends Component {
  render() {
    return [                                                    | Возвращение массива
      <div key="displayMap" className="displayMap">             | элементов от рендера
```

```
        <div
          className="map"
          ref={node => {
            this.mapNode = node;
          }}
        >
        </div>
      </div>
    ];
  }
}
```

DOM-элемент, который Mapbox будет использовать для создания карты

Хорошее начало обеспечения поддержки карт в React. В дальнейшем для разработки карты нужно применить API Mapbox. Вы создадите метод, который будет задействовать ссылку, сохраненную в классе. Нужно также настроить некоторые свойства и состояние по умолчанию и указать, что отображается определенная область карты в масштабе, чтобы она не запускалась, показывая весь мир. Вы запишете в компоненте несколько фрагментов состояния, включая сведения о том, загружена ли карта, и некоторую информацию о местоположении (широту, долготу и название места). Обратите внимание на то, что вопрос о взаимодействии с другой библиотекой JavaScript через React довольно тривиален. Самое сложное заключается в использовании ссылок, но, не учитывая это, библиотеки довольно эффективно могут работать совместно. В листинге 6.4 показано, как настроить компонент `DisplayMap`.

Листинг 6.4. Создание карты с помощью Mapbox (src/components/map/DisplayMap.js)

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

export default class DisplayMap extends Component {
  constructor(props) {
    super(props);
    this.state = {
      mapLoaded: false,
      location: {
        lat: props.location.lat,
        lng: props.location.lng,
        name: props.location.name
      }
    };
    this.ensureMapExists = this.ensureMapExists.bind(this);
  }
  static propTypes = {
    location: PropTypes.shape({
      lat: PropTypes.number,
      lng: PropTypes.number,
      name: PropTypes.string
    }),
    displayOnly: PropTypes.bool
  };
```

Настройка начального состояния

Привязка метода класса ensureMapExists

```
  static defaultProps = {
    displayOnly: true,
    location: {
      lat: 34.1535641,
      lng: -118.1428115,
      name: null
    }
  };
  componentDidMount() {
    this.L = window.L;
    if (this.state.location.lng && this.state.location.lat) {
      this.ensureMapExists();
    }
  }
  ensureMapExists() {
    if (this.state.mapLoaded) return;
    this.map = this.L.mapbox.map(this.mapNode,
      'mapbox.streets', {
      zoomControl: false,
      scrollWheelZoom: false
    });
    this.map.setView(this.L.latLng(this.state.location.lat,
    this.state.location.lng), 12);

    this.setState(() => ({ mapLoaded: true }));
  }
  render() {
    return [
      <div key="displayMap" className="displayMap">
        <div
          className="map"
          ref={node => {
            this.mapNode = node;
          }}
        >
        </div>
      </div>
    ];
  }
}
```

В Mapbox используется библиотека под названием Leaflet (отсюда «L»)

Проверка того, есть ли у карты информация о местоположении. Если да, карта настраивается

Проверка того, что вы случайно не пересоздаете карту, если она уже загружена

Создание новой карты с помощью Mapbox и сохранение ссылки на нее в компоненте (отключаются ненужные функции карты)

Обновление состояния, чтобы уведомить о загрузке карты

Установка вида карты на основе значений широты и долготы, полученных компонентом

Теперь ваш компонент должен отображать приемлемую карту. Помните, однако, что вы хотите создать компонент карты, в котором можете указать конкретное местоположение, и обновить его для пользователя, когда он выберет новое местоположение. Вам нужно еще немного поработать, чтобы реализовать такие возможности: выполнить методы для добавления маркера на карту, обновления положения карты и обеспечения правильного обновления. В листинге 6.5 показано, как добавить эти методы к коду компонента.

Возможно, когда вы добавляли каждый из методов к компоненту, то обнаружили такую последовательность: сделать что-то со сторонней библиотекой, сообщить об этом React, повторить. Обычно интеграция со сторонними библиотеками основы-

вается на опыте. Вы, как правило, хотите найти точку интеграции, в которой можно получить данные из библиотеки или использовать ее API, чтобы реализовать функционал, но в пределах React. Возникает много исключений, где может оказаться невероятно сложно достичь этого, но, по моему опыту, сочетание ссылок React и общей совместимости JavaScript облегчает применение сторонних библиотек (и я надеюсь, что вы придете к тому же мнению, работая с будущими React-приложениями).

Листинг 6.5. Динамическая карта (src/components/map/DisplayMap.js)

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

export default class DisplayMap extends Component {
  constructor(props) {
    super(props);
    this.state = {
      mapLoaded: false,
      location: {
        lat: props.location.lat,
        lng: props.location.lng,
        name: props.location.name
      }
    };
    this.ensureMapExists = this.ensureMapExists.bind(this);
    this.updateMapPosition = this.updateMapPosition.bind(this);
  }
  //...
  componentDidUpdate() {
    if (this.map && !this.props.displayOnly) {
      this.map.invalidateSize(false);
    }
  }
  componentWillReceiveProps(nextProps) {
    if (nextProps.location) {
      const locationsAreEqual =
        Object.keys(nextProps.location).every(
        k => nextProps.location[k] === this.props.location[k]
      );
      if (!locationsAreEqual) {
        this.updateMapPosition(nextProps.location);
      }
    }
  }
  //...
  ensureMapExists() {
    if (this.state.mapLoaded) return;
    this.map = this.L.mapbox.map(this.mapNode, 'mapbox.streets', {
      zoomControl: false,
      scrollWheelZoom: false
    });
    this.map.setView(this.L.latLng(this.state.location.lat,
    this.state.location.lng), 12);
```

Привязка методов класса

Mapbox аннулирует размер карты, чтобы она отображалась корректно при скрытии/показе

Когда местоположение для отображения изменяется, нужно отреагировать соответствующим образом

Если доступно местоположение, проверка текущего и предыдущего местоположения, чтобы узнать, одинаковы ли свойства. Если нет, можно обновить карту

```
    this.addMarker(this.state.location.lat, this.state.location.lng);
    this.setState(() => ({ mapLoaded: true }));
  }
  updateMapPosition(location) {
    const { lat, lng } = location;
    this.map.setView(this.L.latLng(lat, lng));
    this.addMarker(lat, lng);
    this.setState(() => ({ location }));
  }
  addMarker(lat, lng) {
    if (this.marker) {
      return this.marker.setLatLng(this.L.latLng(lat, lng));
    }
    this.marker = this.L.marker([lat, lng], {
      icon: this.L.mapbox.marker.icon({
        'marker-color': '#4469af'
      })
    });
    this.marker.addTo(this.map);
  }
  render() {
    return [
      <div key="displayMap" className="displayMap">
        <div
          className="map"
          ref={node => {
            this.mapNode = node;
          }}
        >
        </div>
      </div>
    ];
  }
}
```

Добавление маркера на карту при создании

Обновление представления карты и состояния компонента

Обновление существующего маркера, а не создание его каждый раз заново

Создание маркера и добавление его на карту

Есть по крайней мере одно улучшение, которое вы можете добавить в свой компонент. Mapbox позволяет создавать статические изображения карт на основе географической информации. Это оказывается полезным тогда, когда не требуется загружать интерактивную карту. Вы добавите эту функцию в качестве резервной возможности, чтобы пользователи могли сразу увидеть карту. Она пригодится в главе 12, когда вы займетесь визуализацией (рендерингом) на стороне сервера. Сервер будет генерировать разметку, которая не будет вызывать никаких связанных с монтированием методов, поэтому пользователи все равно смогут видеть местоположение сообщений даже до того, как приложение будет полностью загружено.

Вам также необходимо добавить небольшой элемент пользовательского интерфейса к компоненту карты, чтобы она отображала название геопозиции в режиме «только отображение». Ранее упоминалось, что вы добавляли сестринский элемент к основным элементам и именно поэтому возвращали массив элементов. Здесь вы создадите этот небольшой фрагмент разметки. В листинге 6.6 показано, как добавить в компонент запасные варианты отображения и названия местоположения.

Листинг 6.6. Добавление резервного изображения карты (src/components/map/DisplayMap.js)

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

export default class DisplayMap extends Component {
  constructor(props) {
    super(props);
    this.state = {
      mapLoaded: false,
      location: {
        lat: props.location.lat,
        lng: props.location.lng,
        name: props.location.name
      }
    };
    this.ensureMapExists = this.ensureMapExists.bind(this);
    this.updateMapPosition = this.updateMapPosition.bind(this);
    this.generateStaticMapImage = this.generateStaticMapImage.bind(this);
  }
  //...
  generateStaticMapImage(lat, lng) {
    return 'https://api.mapbox.com/styles/v1/mapbox/streetsv10/
      static/${lat},${lng},12,0,0/600x175?access_token=${process
      .env.MAPBOX_API_TOKEN}';
  }
  render() {
    return [
      <div key="displayMap" className="displayMap">
        <div
          className="map"
          ref={node => {
            this.mapNode = node;
          }}
        >
          {!this.state.mapLoaded && (
            <img
              className="map"
              src={this.generateStaticMapImage(
                this.state.location.lat,
                this.state.location.lng
              )}
              alt={this.state.location.name}
            />
          )}
        </div>
      </div>,
      this.props.displayOnly && (
        <div key="location-description"
          className="locationdescription">
          <i className="location-icon fa fa-location-arrow" />
          <span className="location-name">{
            this.state.location.name}</span>
        </div>
      )
    ];
  }
}
```

Привязка метода класса

Использование широты и долготы для создания URL-адреса изображения из Mapbox

Вывод изображения местоположения

Если выбран режим «только отображение», указываются название и индикатор местоположения

6.2.2. Создание компонента LocationTypeAhead

Вы научились показывать в своем приложении карты, но не сделали ничего для их разработки. Для поддержки этой функции вам необходимо выполнить еще один компонент — *опережение местоположения*. В следующем подразделе вы будете использовать его в компоненте CreatePost, с которым работали, чтобы пользователи могли искать местоположение. Этот компонент будет применять браузерный API геолокации, а также API Mapbox для поиска местоположения.

Начнем с файла `src/components/map/LocationTypeAhead.js`. На рис. 6.2 показан компонент опережения, который вы выполните в этом подразделе.

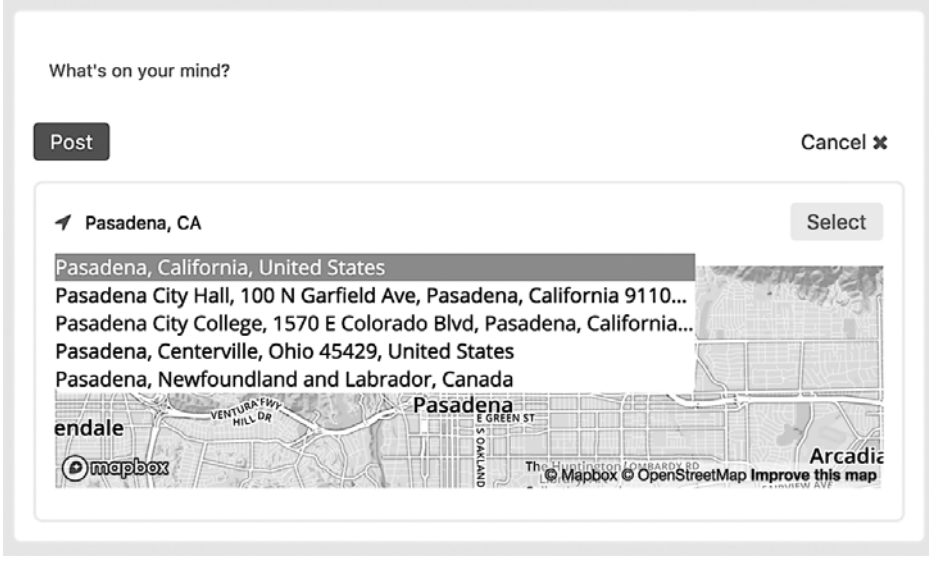


Рис. 6.2. Компонент опережения местоположения, который можно использовать вместе с компонентом карты, чтобы пользователь мог добавлять местоположение в свои сообщения

Вот основные функциональные возможности, которыми будет обладать компонент к моменту завершения работы над ним:

- отображение списка мест для выбора пользователем;
- передача выбранного места родительскому компоненту для использования;
- применение API Mapbox и геолокации, чтобы пользователи могли выбрать свое местоположение или выполнить поиск по адресу.

Затем вы создадите каркас компонента. В листинге 6.7 показан первый эскиз. Вы снова воспользуетесь Mapbox, но на этот раз с другим набором API. В предыдущем подразделе вы задействовали API отображения карты, а здесь примените набор API Mapbox, которые позволят пользователям делать *обратное геокодирование*, — причудливый способ поиска реального местоположения по тексту. Модуль Mapbox уже установлен в проекте и будет работать с тем же самым публичным ключом

Mapbox. Если вы ранее добавили в API собственный ключ, он должен быть указан в конфигурации приложения.

Упражнение 6.1. Альтернативы Mapbox

В данной главе вы использовали Mapbox, но есть и другие картографические библиотеки, такие как Google Maps. Как бы вы отключили Mapbox, чтобы работать с Google Maps? Что вам нужно было бы сделать иначе?

Листинг 6.7. Начало компонента LocationTypeAhead

```
import React, { Component } from 'react';
import PropTypes from 'prop-types';
import MapBox from 'mapbox';

export default class LocationTypeAhead extends Component {
  static propTypes = {
    onLocationUpdate: PropTypes.func.isRequired,
    onLocationSelect: PropTypes.func.isRequired
  };
  constructor(props) {
    super(props);
    this.state = {
      text: '',
      locations: [],
      selectedLocation: null
    };
    this.mapbox = new MapBox(process.env.MAPBOX_API_TOKEN);
  }
  render() {
    return [
      <div key="location-typeahead"
        className="location-typeahead">
        <i className="fa fa-location-arrow"
        onClick={this.attemptGeoLocation} />
        <input
          onChange={this.handleSearchChange}
          type="text"
          placeholder="Enter a location..."
          value={this.state.text}
        />
        <button
          disabled={!this.state.selectedLocation}
          onClick={this.handleSelectLocation}
          className="open"
        >
          Select
        </button>
      </div>
    ];
  }
}
```

Импорт Mapbox

Использование двух методов: одного для обновления местоположения, другого — для его выбора

Установка начального состояния

Создание экземпляра клиента Mapbox

Возвращение массива элементов, который станет разметкой для компонента опережения местоположения. Вам нужно реализовать все методы, на которые ссылаются обработчики событий (onChange, onClick и т. д.)

Теперь вы можете наполнять методы, на которые ссылались в методе `render` компонента. Обратите внимание на то, что вам требуются способ обработки изменений в тексте поиска, кнопка для реализации возможности выбора местоположения и значок, который позволит пользователю указать свое местоположение. Я расскажу об этой функциональности чуть позже, а сейчас вам нужны методы, которые дадут пользователю возможность искать местоположение по названиям и выбирать его. В листинге 6.8 показано, как добавить данные методы. Откуда берется это местоположение? Вы будете использовать API Mapbox для поиска местоположения на основе ввода, выполненного пользователем, и применять эти результаты для отображения их адресов. Это лишь один способ работы с Mapbox. Можно сделать и обратное — ввести координаты и превратить их в адрес. Вы реализуете эту возможность в листинге 6.8 для работы с API геолокации.

Листинг 6.8. Поиск местоположения (src/components/map/LocationTypeAhead.js)

```
//...
constructor(props) {
  super(props);
  this.state = {
    text: '',
    locations: [],
    selectedLocation: null
  };
  this.mapbox = new MapBox(process.env.MAPBOX_API_TOKEN);
  this.handleLocationUpdate = this.handleLocationUpdate.bind(this);
  this.handleSearchChange = this.handleSearchChange.bind(this);
  this.handleSelectLocation = this.handleSelectLocation.bind(this);
  this.resetSearch = this.resetSearch.bind(this);
}
componentWillUnmount() {
  this.resetSearch();
}
handleLocationUpdate(location) {
  this.setState(() => {
    return {
      text: location.name,
      locations: [],
      selectedLocation: location
    };
  });
  this.props.onLocationUpdate(location);
}
handleSearchChange(e) {
  const text = e.target.value;
  this.setState(() => ({ text }));
  if (!text) return;
  this.mapbox.geocodeForward(text, {}).then(loc => {
    if (!loc.entity.features || !loc.entity.features.length) {
      return;
    }
```

Привязка методов класса

Когда компонент размонтируется, поиск сбрасывается

Когда местоположение выбрано, обновляется локальное состояние компонента

В то же время местоположение передается предку через обратный вызов свойств

Извлекаем текст из события, которое получаем, когда пользователь что-то вводит в поле поиска

Используется клиент Mapbox для поиска местоположения с учетом ввода пользователя

Бездействие, если нет результатов

```
    const locations = loc.entity.features.map(feature => {
      const [lng, lat] = feature.center;
      return {
        name: feature.place_name,
        lat,
        lng
      };
    });
  this.setState(() => ({ locations }));
  });
}
resetSearch() {
  this.setState(() => {
    return {
      text: '',
      locations: [],
      selectedLocation: null
    };
  });
}
handleSelectLocation() {
  this.props.onLocationSelect(this.state.selectedLocation);
}
//...
```

Преобразование результатов Mapbox в формат, который проще использовать в компоненте

Обновление состояния с новым местоположением

Разрешение сброса состояния компонента (см. componentWillUnmount)

Когда выбрано местоположение, данная локация передается вверх

Теперь вы хотите, чтобы пользователь мог выбрать свое местоположение для сообщения. Для этого применяйте API геолокации браузера. Не волнуйтесь, если раньше не работали с API геолокации. Долго это была специфическая функция, и ее можно было использовать только в определенных браузерах. Теперь она получила широкое распространение и стала более полезной.

API геолокации работает так: вы спрашиваете у пользователя, можете ли использовать его местоположение в своем приложении. Сейчас почти все браузеры поддерживают API геолокации (caniuse.com/#feat=geolocation), поэтому можно воспользоваться им и разрешить пользователю выбирать текущее местоположение для сообщения. Обратите внимание: API геолокации можно применять только в защищенных средах, поэтому, если вы попытаетесь развернуть Letters Social на незащищенном хосте, прием не сработает.

Вам нужно снова задействовать API Mapbox, поскольку все, на что способен API геолокации, — это определение координат. Помните, как вы задействовали текст пользователя для поиска местоположения в Mapbox? Можете сделать обратное: задать координаты Mapbox и получить обратно соответствующие адреса. В листинге 6.9 показано, как использовать API геолокации и Mapbox, чтобы пользователь мог выбрать свое местоположение для сообщения.

Компонент позволяет искать местоположение в Mapbox и разрешает пользователю выбирать свое местоположение с помощью API геолокации. Но он пока ничего не показывает ему, и это следует исправить. Нужно использовать результаты геолокации, чтобы пользователь мог выбрать один из них, как показано в листинге 6.10.

Листинг 6.9. Добавление геолокации (src/components/map/LocationTypeAhead.js)

```
constructor(props) {
  super(props);
  this.state = {
    text: '',
    locations: [],
    selectedLocation: null
  };
  this.mapbox = new MapBox(process.env.MAPBOX_API_TOKEN);
  this.attemptGeoLocation = this.attemptGeoLocation.bind(this);
  this.handleLocationUpdate = this.handleLocationUpdate.bind(this);
  this.handleSearchChange = this.handleSearchChange.bind(this);
  this.handleSelectLocation = this.handleSelectLocation.bind(this);
  this.resetSearch = this.resetSearch.bind(this);
}
//...
attemptGeoLocation() {
  if ('geolocation' in navigator) {
    navigator.geolocation.getCurrentPosition(
      ({ coords }) => {
        const { latitude, longitude } = coords;
        this.mapbox.geocodeReverse({ latitude, longitude },
        {}).then(loc => {
          if (!loc.entity.features ||
          !loc.entity.features.length) {
            return;
          }
          const feature = loc.entity.features[0];
          const [lng, lat] = feature.center;
          const currentLocation = {
            name: feature.place_name,
            lat,
            lng
          };
          this.setState(() => ({
            locations: [currentLocation],
            selectedLocation: currentLocation,
            text: currentLocation.name
          }));
          this.handleLocationUpdate(currentLocation);
        });
      },
      null,
      {
        enableHighAccuracy: true,
        timeout: 5000,
        maximumAge: 0
      }
    );
  }
}
//...
```

Привязка метода класса

Проверка того, поддерживает ли браузер геолокацию

Получение текущей позиции пользовательского устройства

Возвращение координат, которые вы можете использовать

Применение Mapbox для геокодирования координат и преждевременного возврата, если ничего не найдено

Получение первого (ближайшего) свойства

Выдача широты и долготы

Создание полезной нагрузки местоположения и обновления состояния компонента

Вызов свойства handleLocationUpdate с новым местоположением

Параметры, передаваемые API геолокации

Листинг 6.10. Отображение результатов для пользователя (src/components/map/LocationTypeAhead.js)

```
//...
render() {
  return [
    <div key="location-typeahead" className="location-typeahead">
      <i className="fa fa-location-arrow"
      onClick={this.attemptGeoLocation} />
      <input
        onChange={this.handleSearchChange}
        type="text"
        placeholder="Enter a location..."
        value={this.state.text}
      />
      <button
        disabled={!this.state.selectedLocation}
        onClick={this.handleSelectLocation}
        className="open"
      >
        Select
      </button>
    </div>,
    this.state.text.length && this.state.locations.length ? (
      <div key="location-typeahead-results"
        className="locationtypeahead-results">
        {this.state.locations.map(location => {
          return (
            <div
              onClick={e => {
                e.preventDefault();
                this.handleLocationUpdate(location);
              }}
              key={location.name}
              className="result"
            >
              {location.name}
            </div>
          );
        })}
      </div>
    ) : null
  ];
}
//...
```

Если есть поисковый запрос и получены соответствующие результаты, вывод результатов

Установка соответствия по местам, полученным от Mapbox

Если пользователь щелкает кнопкой мыши на позиции, установить выбранное местоположение

Не забудьте задать ключи компонентам, которые перебираете

Вывод географического названия места

Если нет местоположения и поискового запроса, ничего не делать

6.2.3. Обновление CreatePost и добавление карт в сообщения

Теперь, когда вы разработали компоненты `LocationTypeAhead` и `DisplayMap`, можете интегрировать их в компонент `CreatePost`. Это свяжет созданную функциональность и позволит пользователю писать сообщения, в которых указывается местоположение. Помните, как компонент `CreatePost` передает свои данные обратно

в родительский компонент для фактического создания сообщения? Вы будете делать то же самое с компонентами опережения местоположения и отображения карты, но из `CreatePost`. Они станут работать вместе, но не будут настолько привязаны друг к другу, чтобы вы не смогли их перемещать или использовать в другом месте.

Необходимо обновить компонент `CreatePost` для работы с ранее созданными компонентами `LocationTypeAhead` и `DisplayMap`, которые хранят, передают и получают местоположение соответственно. Вы будете отслеживать местоположение в компоненте `CreatePost` и задействовать два недавно разработанных компонента в качестве источника и адресата данных о местоположении. В листинге 6.11 показано, как создать методы, необходимые для включения местоположения в сообщения.

Листинг 6.11. Обработка местоположения в CreatePost (src/components/post/Create.js)

```
constructor(props) {
  super(props);
  this.initialState = {
    content: '',
    valid: false,
    showLocationPicker: false,
    location: {
      lat: 34.1535641,
      lng: -118.1428115,
      name: null
    },
    locationSelected: false
  };
  this.state = this.initialState;
  this.filter = new Filter();
  this.handlePostChange = this.handlePostChange.bind(this);
  this.handleRemoveLocation = this.handleRemoveLocation.bind(this);
  this.handleSubmit = this.handleSubmit.bind(this);
  this.handleToggleLocation = this.handleToggleLocation.bind(this);
  this.onLocationSelect = this.onLocationSelect.bind(this);
  this.onLocationUpdate = this.onLocationUpdate.bind(this);
}
//...
handleRemoveLocation() {
  this.setState(() => ({
    locationSelected: false,
    location: this.initialState.location
  }));
}
handleSubmit() {
  if (!this.state.valid) {
    return;
  }
  const newPost = {
    content: this.state.content
  };
  if (this.state.locationSelected) {
    newPost.location = this.state.location;
  }
```

Добавление ключей в состояние, чтобы можно было отслеживать местоположение и связанные данные. Настройка некоторых данных местоположения по умолчанию

Связывание методов класса

Разрешение пользователю удалять местоположение из своего сообщения

При отправке сообщения добавление в полезную нагрузку местоположения, если оно имеется

```
  this.props.onSubmit(newPost);
  this.setState(() => ({
    content: '',
    valid: false,
    showLocationPicker: false,
    location: this.initialState.location,
    locationSelected: false
  }));
}
onLocationUpdate(location) {
  this.setState(() => ({ location }));
}
onLocationSelect(location) {
  this.setState(() => ({
    location,
    showLocationPicker: false,
    locationSelected: true
  }));
}
handleToggleLocation(e) {
  e.preventDefault();
  this.setState(state => ({ showLocationPicker:
  !state.showLocationPicker }));
}
//...
```

Обработка обновления местоположения из компонента LocationTypeAhead

Переключение отображения выбора местоположения

Теперь компонент `CreatePost` позволяет работать с локациями, поэтому нужно добавить пользовательский интерфейс, чтобы реализовать эту возможность. Как только вы создадите пользовательский интерфейс для добавления местоположения, то увидите, что метод `render` стал немного запутанным. Это неплохо, и разметка не настолько сложна, чтобы пришлось реорганизовывать код (я работал с методами `render`, код которых состоит из сотен строк), но это хорошая возможность изучить другую технику рендеринга в компоненте React, которую я называю *субрендерингом*.

Упражнение 6.2 Использование ссылок повсюду

В этой главе мы потратили некоторое время на изучение того, как применять ссылки в React. Можете ли вы назвать другие библиотеки или ситуации, в которых ссылки будут полезны? Работали ли вы над проектами, в которых нужно использовать ссылки для интеграции с React?

Метод *субрендеринга* разделяет часть метода `render` на метод класса в компоненте (или функцию в любой позиции), а затем вызывает его в выражении JSX в основном методе `render`. Можете пользоваться этим приемом, если нужно разделить сложный метод `render`, изолировать логику для определенной части визуализированного пользовательского интерфейса или в иных случаях. Вероятно, могут возникнуть и другие ситуации, когда этот прием применим, но ключевым является то, что

можно разделить рендеринг на несколько частей, которые могут не быть другими компонентами. Листинг 6.12 иллюстрирует разделение кода метода `render` на более мелкие части.

Листинг 6.12. Добавление метода субрендеринга (src/components/post/Create.js)

```
  constructor(props) {
    //...
    this.renderLocationControls = this.renderLocationControls.bind(this);
  }
  renderLocationControls() {
    return (
      <div className="controls">
        <button onClick={this.handleSubmit}>Post</button>
        {this.state.location && this.state.locationSelected ? (
          <button onClick={this.handleRemoveLocation}
          className="open location-indicator">
            <i className="fa-location-arrow fa" />
            <small>{this.state.location.name}</small>
          </button>
        ) : (
          <button onClick={this.handleToggleLocation}
          className="open">
            {this.state.showLocationPicker ? 'Cancel' :
            'Add location'}{' '}
            <i
              className={classnames('fa', {
                'fa-map-o': !this.state.showLocationPicker,
                'fa-times': this.state.showLocationPicker
              })}
            />
          </button>
        )}
      </div>
    );
  }
  render() {
    return (
      <div className="create-post">
        <textarea
          value={this.state.content}
          onChange={this.handlePostChange}
          placeholder="What's on your mind?"
        />
        {this.renderLocationControls()}
        <div
          className="location-picker"
          style={{ display: this.state.showLocationPicker ? 'block'
          : 'none' }}
```

Привязка метода класса в конструкторе

Если местоположение выбрано, отображение кнопки, которая позволяет пользователям удалять свое местоположение

Привязка метода removeLocation и вывод текущего местоположения

Отображение кнопки, переключающей компоненты выбора местоположения

Отображение корректного текста и использование правильного метода привязки, основанного на состоянии местоположения

Вызов метода субрендеринга

Отображение или скрытие компонентов выбора местоположения в зависимости от состояния

```
        >
          {!this.state.locationSelected && [
            <LocationTypeAhead
              key="LocationTypeAhead"
              onLocationSelect={this.onLocationSelect}
              onLocationUpdate={this.onLocationUpdate}
            />,
            <DisplayMap
              key="DisplayMap"
              displayOnly={false}
              location={this.state.location}
              onLocationSelect={this.onLocationSelect}
              onLocationUpdate={this.onLocationUpdate}
            />
          ]}
        </div>
      </div>
    );
  }
```

Отображение компонентов выбора местоположения, если местоположение не выбрано

Наконец, нужно добавить карты к сообщениям, в которых указано местоположение. Вы уже создали компонент `DisplayMap` и убедились, что он может работать в режиме только отображения, поэтому остается лишь включить его в компонент `Post`. В листинге 6.13 показано, как это сделать.

Листинг 6.13. Добавление карт в сообщения (src/components/post/Post.js)

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

import * as API from '../../shared/http';
import Content from './Content';
import Image from './Image';
import Link from './Link';
import PostActionSection from './PostActionSection';
import Comments from '../comment/Comments';
import DisplayMap from '../map/DisplayMap';
import UserHeader from '../post/UserHeader';
import Loader from '../Loader';

export class Post extends Component {
  static propTypes = {
    post: PropTypes.object
  };
  //...
  render() {
    if (!this.state.post) {
      return <Loader />;
    }
    return (
      <div className="post">
        <UserHeader date={this.state.post.date}
```

Импорт компонента DisplayMap

```
          user={this.state.post.user} />
        <Content post={this.state.post} />
        <Image post={this.state.post} />
        <Link link={this.state.post.link} />
        {this.state.post.location && <DisplayMap
          location={this.state.post.location} />}
        <PostActionSection showComments={this.state.showComments} />
        <Comments
          comments={this.state.comments}
          show={this.state.showComments}
          post={this.state.post}
          handleSubmit={this.createComment}
          user={this.props.user}
        />
      </div>
    );
  }
}

export default Post;
```

Если сообщение имеет связанное с ним местоположение, отображение его и активизация режима displayOnly

При этом вы реализовали возможность добавлять местоположение в сообщения и отображать его для пользователей. Ваши инвесторы, несомненно, будут счастливы и впечатлены такой возможностью!

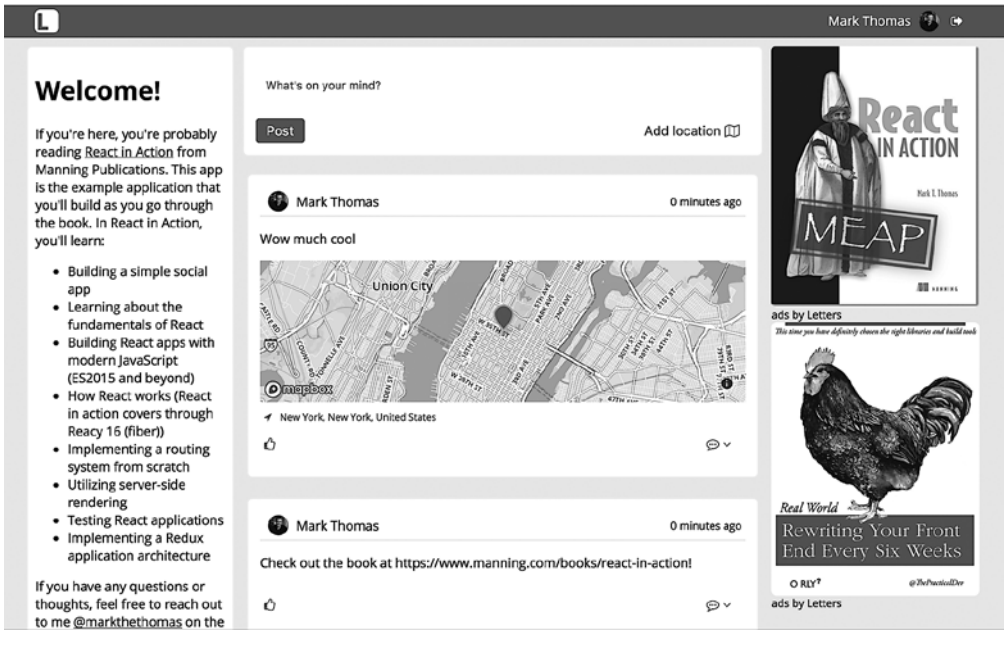


Рис. 6.3. Итоговый вариант программы из этой главы. Пользователи могут создавать сообщения и добавлять к ним местоположение

6.3. Резюме

Вот что вы узнали в этой главе.

- В React `ref` — это ссылка на базовый DOM-элемент. Ссылки могут быть полезны, когда вам требуется обходной маневр и нужно работать с библиотеками, обрабатывающими DOM вне React.
- Компоненты могут быть контролируемыми или неконтролируемыми. Контролируемые компоненты позволяют вам следить за состоянием компонента и включают полный цикл прослушивания и затем установки значения ввода. Неконтролируемые компоненты поддерживают собственное внутреннее состояние и не обеспечивают наблюдаемости или контроля.
- Интегрировать компоненты React со сторонними библиотеками, которые также применяют DOM, зачастую возможно с помощью ссылок. Последние могут выступать в качестве обходного маневра, когда нужно взаимодействовать с DOM-элементами.

В следующей главе вы усложните проект и разработаете базовую маршрутизацию для своего приложения, чтобы появилась возможность использовать несколько страниц.

7 Маршрутизация в React

- Более продвинутый дизайн и использование компонентов.
- Создание многостраничных React-приложений с помощью маршрутизации.
- Разработка роутера с нуля с помощью React.

В этой главе вы повысите надежность и масштабируемость своего приложения, добавив маршрутизацию. *Маршрутизация* означает, что пользователи смогут перемещаться по различным разделам приложения с помощью URL-адресов. До сих пор приложение состояло лишь из одной страницы, что затрудняло развитие, когда вы добавляли разделы контента. Крупные приложения будут особенно страдать от переполнения без маршрутизации или другого механизма, обеспечивающего управляемую иерархию. Мы рассмотрим, как решить эту проблему в приложении с помощью React. Вы создадите простой роутер с нуля, чтобы лучше понять, как выполнять маршрутизацию в приложениях React.

Получение исходного кода

Как и прежде, вы можете получить исходный код примеров из этой главы, перейдя в репозиторий GitHub по адресу github.com/react-in-action/letters-social. Если вы хотите начать работу здесь самостоятельно с нуля, возьмите исходный код примеров из глав 5 и 6 (если изучили их и сами выполнили примеры) или обратитесь к ветви, относящейся к данной главе (chapter-7-8).

Помните, что каждая ветвь содержит итоговый код главы (например, chapter-7-8 содержит код, получаемый в конце глав 7 и 8). Вы можете выполнить в оболочке командной строки одну из следующих команд по своему выбору, чтобы получить код примеров из текущей главы. Если репозитория вообще нет, выполните такую команду:

```
git clone git@github.com:react-in-action/letters-social.git
```

Если у вас уже клонирован репозиторий, то следующую:

```
git checkout chapter-7-8
```

Возможно, вы перешли сюда из другой главы, поэтому стоит проверить, установлены ли у вас все нужные зависимости, с помощью следующей команды:

```
npm install
```

7.1. Что такое маршрутизация

Чтобы успешно настроить маршрутизацию, вы должны иметь представление о том, что это такое. Маршрутизация — ключевой компонент всех веб-сайтов и веб-приложений. Она играет центральную роль на простейших статических HTML-страницах и в самых сложных React-приложениях в Интернете. Маршрутизация вступает в игру практически каждый раз, когда вы хотите сопоставить URL-адрес с действием. Большинство приложений заполнены URL-ссылками, потому что ссылки — это де-факто способ перемещения по Сети. Подумайте о том, насколько эффективна система поиска, которой стали URL-адреса — они используются почти повсюду. Почему они так полезны для поиска в Интернете? Возможно, потому, что мы привыкли к системам маршрутизации, таким как адреса, и даже если URL-адреса не требуют указателей движения от поворота до поворота, они помогают найти то, что мы ищем, — в данном случае приложения или ресурсы вместо местоположения.

ОПРЕДЕЛЕНИЕ

> Маршрутизация имеет много разных значений и реализаций. Для нас это система навигации по ресурсам. Вероятно, маршрутизация вам привычна и вы активно применяете ее в веб-разработке. Если вы запустили браузер, значит, знакомы с маршрутизацией, поскольку здесь не обойтись без URL-адресов и ресурсов в браузере (пути к изображениям, сценариям и т. д.). На сервере маршрутизация может быть сосредоточена на сопоставлении входных маршрутов запросов (например, ifelse.io/react-ecosystem) с ресурсами из базы данных. Вы изучаете, как использовать React, поэтому в этой книге маршрутизация обычно означает соответствие компонентов (ресурсов, которые нужны людям) и URL (способа сообщить системе, чего они хотят).

Маршрутизация — важный компонент веб-приложений. Предположим, вы хотите разработать веб-приложение, в котором пользователи могут создавать собственные страницы сбора средств на благотворительность. В этом случае маршрутизация понадобится вам по нескольким причинам.

- Люди могут постить внешние ссылки на веб-приложение. URL-адреса, указывающие на постоянные ресурсы, должны быть долговечными и сохранять целостность структуры с течением времени.
- Публичные страницы сбора средств должны быть доступны всем, поэтому вам нужен URL-адрес, который направит их на нужную страницу.
- Для обеспечения названных условий потребуется администрирование. Пользователи должны иметь возможность двигаться вперед и назад по своей истории просмотров.
- Разным частям сайта понадобятся собственные URL-адреса, чтобы можно было легко перенаправить людей в нужный раздел (например, `/settings`, `/profile`, `/prices` и т. д.).
- Разбиение кода на страницы помогает повысить модульность, поэтому вы можете разделить свое приложение на части. Наряду с динамическим содержимым это способно уменьшить размер приложения, которое должно быть загружено в данной точке.

Маршрутизация в современных клиентских веб-приложениях. Раньше в базовой архитектуре веб-приложения использовался иной подход к маршрутизации. Он включал сервер (думаю, созданный на Python, Ruby или PHP), который генерировал HTML-разметку и отправлял ее в браузер. Пользователь мог заполнить форму какими-то данными, отправить ее обратно на сервер и дожидаться ответа. Это была революция: Интернет стал более мощным, поэтому вы могли изменять данные, а не только просматривать их.

С тех пор веб-службы претерпели значительные изменения в области проектирования и построения. Сейчас фреймворки JavaScript и браузерные технологии достаточно продвинуты для того, чтобы веб-приложения имели более четкое разделение «клиент — сервер». Сервер отправляет клиентское приложение (полностью браузерное), которое затем эффективно «берет верх». Также он отвечает за отправку необработанных данных, обычно в формате JSON. Работа этих двух общих архитектур иллюстрируется и сравнивается на рис. 7.1.

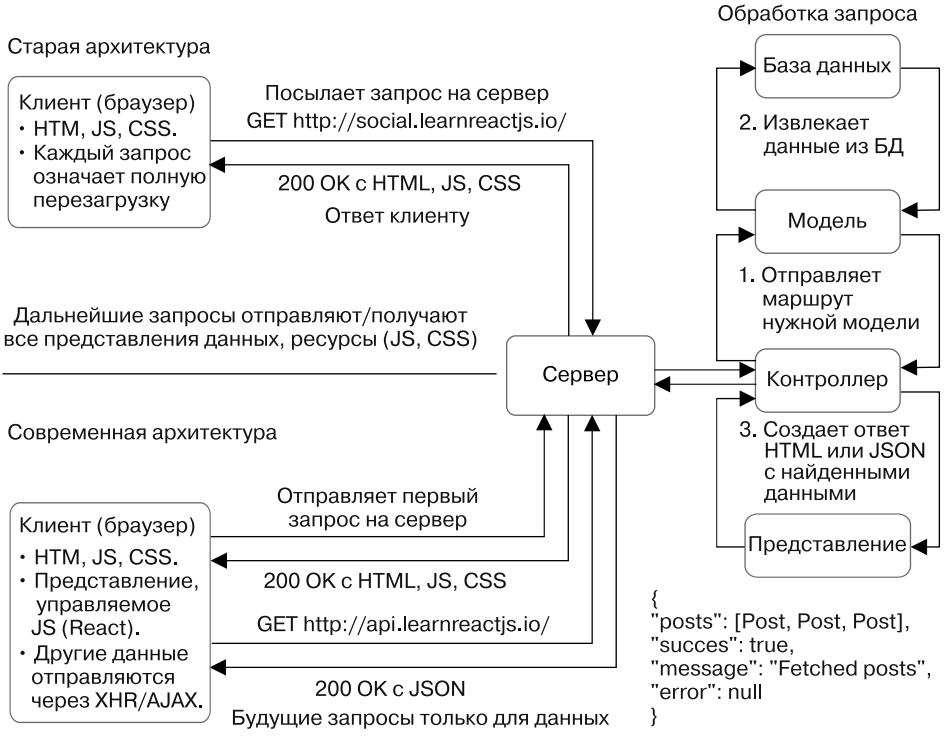


Рис. 7.1. Сравнение слегка устаревших и современных архитектур веб-приложений. Прежде динамическое содержимое генерировалось на сервере. Сервер извлекал данные из базы данных и использовал их для заполнения HTML-представления, которое в дальнейшем отправляли клиенту. Теперь на клиенте больше логики приложения, которая управляется JavaScript (в данном случае React). Сначала сервер отправляет атрибуты HTML, JavaScript и CSS, но после этого клиентское React-приложение берет управление на себя. С этого момента, если пользователь не обновит страницу вручную, серверу придется отправлять только необработанные данные JSON

До сих пор вы применяли современную архитектуру для создания демонстрационного приложения Letters Social. Сервер node.js отправляет код HTML, JavaScript и CSS, который необходим для приложения. Однако, когда все это загрузилось, берет верх React. Дальнейшие запросы данных отправляются на образец сервера API. Но вам не хватает ключевой части этой архитектуры — клиентской маршрутизации.

Упражнение 7.1. Размышления о маршрутизации

Прежде чем мы погрузимся глубже в написание роутера React, задумайтесь на секунду о маршрутизации. С какими примерами маршрутизации вы столкнулись в предыдущих проектах? Где еще используется маршрутизация?

7.2. Создание роутера

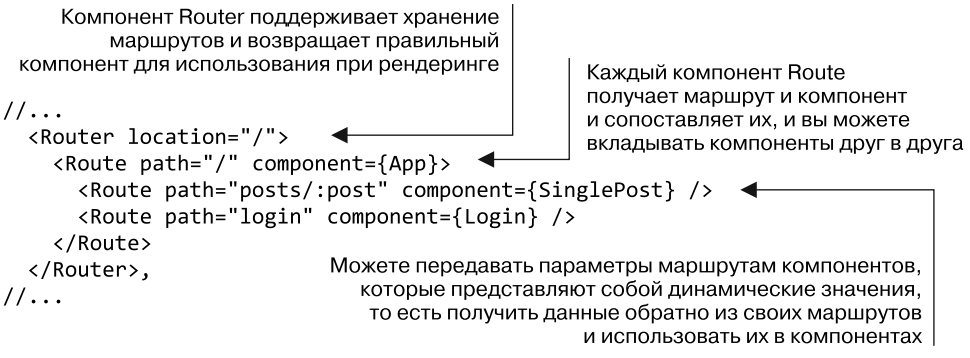
Вы разработаете простой роутер (маршрутизатор) с нуля с помощью компонентов, чтобы лучше понять, как выполнять маршрутизацию в приложениях React.

- Вы создадите два компонента, `Router` и `Route`, которые будут использоваться вместе для выполнения маршрутизации на стороне клиента.
- Компонент `Router` будет содержать компоненты `Route`.
- Каждый компонент `Route` станет представлять URL-путь (`/`, `/posts/123` и т. п.) и сопоставлять компонент с этим URL-адресом. Когда пользователи зайдут на `/`, то увидят соответствующий компонент.
- Компонент `Router` будет выглядеть как обычный компонент `React` (он имеет метод `render` и методы компонента и использует JSX), но позволит сопоставлять компоненты с URL-адресами.
- Компоненты `Route` могут указывать такие параметры, как `/users /: user`, где синтаксис `:user` будет означать значение, переданное компоненту.
- Вы также создадите компонент `Link`, который позволит выполнять навигацию с помощью роутера на стороне клиента.

Если задача еще неясна, не волнуйтесь. Мы проработаем каждый шаг по очереди. Рассмотрим пример работы с роутером.

В листинге 7.1 показан код компонента `Router` в финальной версии. Для упрощения можно представить ситуацию так: роутер с маршрутами, привязанными к компоненту. Маршрутизация не обязательно должна быть иерархической — вы можете выстраивать ее хаотично и вложить ресурсы произвольно, как часто и происходит. Это означает, что маршрутизация относительно легко сопоставляется с семантикой композиции React. Для того, кто только начал изучать React, пример маршрутизации, код которой показан далее, представляется одним из самых простых компонентов.

Листинг 7.1. Готовый код роутера (src/index.js)




```
//...
  <Router location="/">
    <Route path="/" component={App}>
      <Route path="posts/:post" component={SinglePost} />
      <Route path="login" component={Login} />
    </Route>
  </Router>,
//...
```

Такую структуру роутера легко читать и анализировать. Она довольно хорошо зарекомендовала себя в React-приложениях благодаря библиотеке `React Router`. Вы будете придерживаться ее и собирать свой роутер с учетом того же базового API. В процессе работы, обретя вдохновение, мы станем руководствоваться небольшой библиотекой, разработанной Ти Джеем Холовайчуком (TJ Holowaychuk) и названной `react-enroute`. С ее помощью вы можете исследовать маршрутизацию в React без повторного создания библиотеки с открытым исходным кодом, такой как `React Router`.

Мы разобрались в том, что вы будете строить и как оно должно выглядеть в результате, но с чего начать? Начнем с потомков.

7.2.1. Маршрутизация компонентов

Нет, вы не будете привлекать детей для реализации маршрутизации в своем приложении. Вместо этого используйте специальное свойство компонента — `children`. Мы задействовали `children` в предыдущих главах, где оно было частью сигнатуры функции `React.createElement (type, props, children)` и применялось как специальное свойство, с помощью которого можно составлять компоненты.

Раньше вы волновались насчет потомков только с точки зрения ввода: нужно было передать компоненты другому компоненту, чтобы скомпоновать их. Теперь же станете обращаться к потомкам изнутри компонента и применять сами компоненты для настройки своих маршрутов. Можете начать работу с сопоставления компонентов и URL-адресов. Если маршрутизация в веб-разработке — это сопоставление URL-адресов с поведением или представлениями, то маршрутизация в React — это сопоставление URL-адресов с конкретными компонентами.

7.2.2. Создание компонента <Route />

Вы разработаете компонент `Router`, который будет использовать дочерние компоненты для сопоставления URL-адресов с компонентами и их вывода. Если вам сложно понять, как это будет выглядеть, помните: мы будем тщательно анализировать каждый шаг и вам не нужно представлять себе процесс полностью, с самого начала.

В листинге 7.2 рассмотрены два типа компонентов: `Router` и `Route`. Начнем с компонентов `Route`, которые можно задействовать для связывания компонентов с маршрутами. Здесь показано, как написать компонент `Route`. Может показаться, что для маршрутизации этого мало, однако это не так — вполне достаточно. Компонент `Router` выполнит бóльшую часть тяжелой работы, тогда как `Route` станет использоваться в основном как контейнер данных для сопоставления URL-адресов и компонентов.

Листинг 7.2. Создание компонента Route (src/components/router/Route.js)

```
import PropTypes from 'prop-types';
import { Component } from 'react';
import invariant from 'invariant';

class Route extends Component {
  static propTypes = {
    path: PropTypes.string,
    component: PropTypes.oneOfType([PropTypes.element, PropTypes.func]),
  };
  render() {
    return invariant(false, "<Route> elements are for config only and
  shouldn't be rendered");
  }
}

export default Route;
```

Подключение библиотеки invariant, чтобы гарантировать, что компонент Route никогда не будет отображаться, или, если это произойдет, будет показана ошибка

Весь компонент Route — это просто функция, которая возвращает вызов в библиотеку invariant, — если она когда-либо вызывается, возникает ошибка и вы понимаете, что код работает неправильно

Каждый компонент Route принимает маршрут и функцию, поэтому укажите эти свойства с помощью PropTypes

Применяется именованный экспорт, чтобы компонент стал доступным для внешних модулей

Вы, наверное, заметили, что здесь импортируете новую библиотеку, называемую `invariant`. Это простой инструмент, с помощью которого будут выводиться ошибки, если не выполнены определенные условия. Применяя эту библиотеку, вы передаете значение и сообщение. Если значение *ложно* (`null`, `0`, `undefined`, `NaN`, `''` (пустая строка) или `false`), будет вызвана ошибка. Библиотека `invariant` часто используется в React, поэтому, если вы когда-нибудь увидите в консоли инструментов разработчика предупреждение или сообщение об ошибке — что-то вроде «инвариантного нарушения», знайте: она, вероятно, связана с библиотекой. Вы примените ее здесь, чтобы убедиться, что компонент `Route` ничего не отображает.

Правильно — компонент `Route` и не должен ничего отображать. Если он это сделает, инструмент `invariant` выдаст ошибку. Такое поведение может показаться странным. В конце концов, до сих пор вы много работали над рендерингом в своих компонентах. Но это всего лишь способ группировки маршрутов и компонентов таким образом, который React поддерживает и которым можно воспользоваться. Вы будете задействовать компоненты `Route` для хранения свойств и их передачи нужным потомкам. Все станет понятнее, когда вы создадите компонент `Router`, но все же взгляните на рис. 7.2, прежде чем читать дальше.

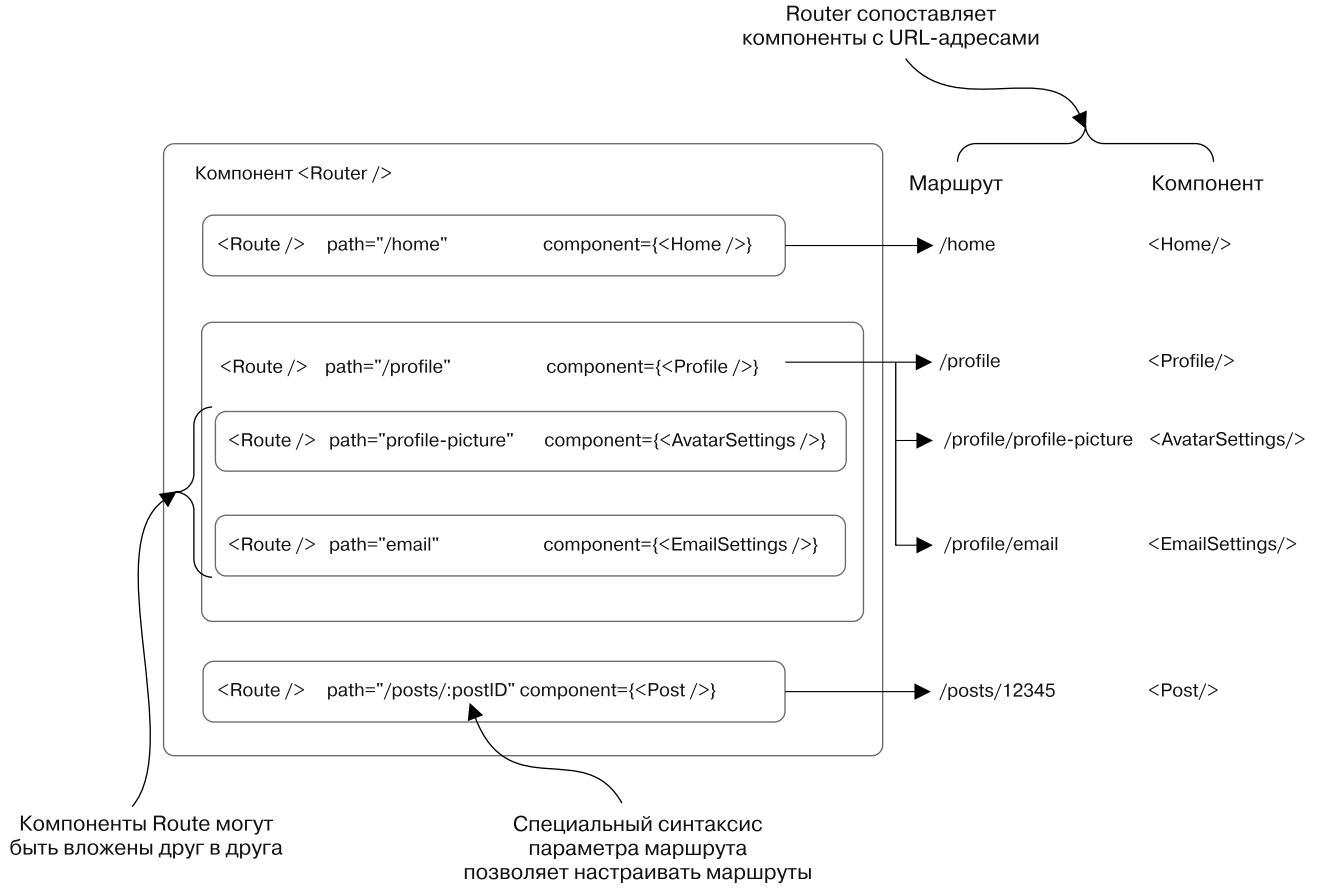


Рис. 7.2. Обзор работы компонентов Route и Router. Router, который вы создадите в следующем подразделе, содержит компоненты Route в качестве дочерних элементов. Каждый из этих компонентов использует два свойства: строку path и компонент. Компонент <Router /> будет применять каждый компонент <Route /> для нахождения соответствия URL-адресу и рендеринга нужного компонента. Поскольку все это является компонентом React, можете передавать свойства роутеру при рендеринге и использовать их как исходное состояние приложения для данных верхнего уровня, таких как пользователь, состояние аутентификации и т. д.

7.2.3. Сборка компонента <Router />

Чтобы начать работу с роутером, нужно вспомнить основы создания компонента. Вы все это уже знаете, хотя в итоге разработаете компонент, совершающий уникальные операции, которых вы еще не видели. Хорошая новость: для написания роутера не нужно делать ничего сверхъестественного. Вы будете работать с компонентами React, добавляя некоторую логику компоненту `Router`, а затем используя его в качестве основного компонента, который ваше приложение рендерит.

Все это кажется пустяком. Вы подумаете: «Хорошо, это компонент. В конце концов, это React, так что все представляется... нормальным?» Я так говорю, потому что это хороший пример чего-то могучего и гибкого, что вы можете делать с «просто» библиотекой React и о чем могли сразу не подумать. Вам не требуются никакие сверхновые инструменты. Нужно лишь найти способ записи сопоставлений URL-адресов и компонентов, а затем способ взаимодействия с правильными API браузера. Теперь постройте этот компонент.

Что насчет React Router?

Возможно, вы слышали о `React Router` прежде, если вообще работали с React. Это один из самых популярных открытых проектов React, предназначенный для маршрутизации в React-приложениях. Вы можете задаться вопросом: почему бы просто не установить `React Router` и не использовать этот API? Вы могли бы это сделать, но, я думаю, упустили бы шанс увидеть, что способны делать что-то неожиданное с компонентами React (например, сопоставлять URL-адреса с компонентами!). Вы приобретете больше умений, разрабатывая что-то самостоятельно, а не просто устанавливая с помощью менеджера npm.

Данный процесс отличается от применяемого в рабочей среде. Как бы это ни было полезно для написания роутера с нуля, основная роль разработчика (почти всегда) заключается в том, чтобы увеличивать ценность компании, и вы можете добиться этого наиболее эффективно, создав или используя хорошо протестированные инструменты, с которыми легко работать.

Учитывая все это, вы и ваша команда, вероятно, предпочтете применять библиотеку `React Router` вместо того, чтобы разрабатывать собственный роутер. Часто лучшее корпоративное решение в области проектирования — выбор хорошо поддерживаемой популярной библиотеки с открытым исходным кодом, которая соответствует вашим потребностям. Когда в главе 12 мы будем обсуждать рендеринг на стороне сервера, вы поменяете свой роутер на инструмент `React Router`, чтобы воспользоваться некоторыми его функциями.

В листинге 7.3 показано, как развернуть компонент `Router`. Здесь нет ничего необычного, кроме свойства `routes`, которое устанавливается в компоненте. Обратите на это внимание, поскольку вам не нужно ничего менять для изменения маршрутов на лету и вы не сохраняете маршруты в локальном состоянии React. Могут возникать

ситуации, когда вы захотите динамически изменять маршруты во время выполнения, например, если пользователь активно настраивает приложение или делает что-то подобное. В таких случаях можете использовать интерфейс `state` компонента. Сейчас необходимости в этом нет, поэтому придерживайтесь маршрутов в компоненте.

Листинг 7.3. Развертывание роутера (src/components/router/Router.js)

```
export default class Router extends Component {
  static propTypes = {
    children: PropTypes.object,
    location: PropTypes.string.isRequired
  };

  constructor(props) {
    super(props);
    this.routes = {};

}

  render() {}
}
```

Компонент Router имеет метод render()

Маршруты будут храниться в компоненте роутера в объекте

Указание PropTypes — роутер получит потомков и локацию для работы

Теперь, когда у вас есть самое необходимое для компонента `Router`, добавьте утилиты, которые станете использовать позже в основных методах компонента.

При работе с маршрутами нужно учитывать следующее. Если вы внимательно посмотрите на листинг 7.2, то, вероятно, заметите, что можно передать свойства `path`, у которых не всегда есть символ `/` перед именем. Это может показаться неважным, но вы должны убедиться, что пользователи роутера могут так сделать. А также должны проконтролировать, что любой двойной слеш (`//`) удаляется, если пользователь добавляет слишком много слешей либо случайно, либо в результате вложенности маршрутов.

Посмотрим, как создать две вспомогательные утилиты для решения этих проблем. Во-первых, вы разработаете утилиту для очистки маршрута. Она будет использовать простое регулярное выражение, заменяющее двойные слеши одинарными. Если вы не знакомы с регулярными выражениями, то найдете много хороших тематических сайтов в Интернете. Такие выражения являются мощным способом сопоставления шаблонов в тексте и становятся ключевыми элементами во многих областях разработки программного обеспечения. В то же время они могут казаться запутанными и трудными для анализа и формирования. К счастью, для поиска и замены любых двойных слешей (`//`) вы применяете простое регулярное выражение. В следующем листинге показано, как реализовать простой метод `cleanPath`. Обратите внимание на то, что очистка строк с помощью регулярных выражений может оказаться сложной, поэтому не думайте, что каждый случай, с которым вы столкнетесь, будет таким же простым.

Мы не будем подробно рассказывать о регулярных выражениях, потому что они заслуживают серьезного и глубокого изучения, достойного отдельной книги, но можем отметить некоторые моменты. Во-первых, базовый синтаксис регулярного выражения в JavaScript — это два слеша с выражением внутри: `/<`*`регулярное выражение`*`>/`. Во-вторых, хотя серия символов `\/\/` выглядит таинственно и, честно

говоря, похожа на букву W, это только два символа / с escape-символами \, поэтому они не интерпретируются как комментарии. Наконец, символ g, добавленный в конец регулярного выражения, представляет собой флаг, указывающий на все вхождения. Чтобы больше узнать о регулярных выражениях, зайдите на сайт regexr.com/3eg8l. Там вы получите подробные сведения о том, что означает каждая из частей регулярного выражения, и сможете попрактиковаться с различными шаблонами.

Листинг 7.4. Добавление утилиты cleanPath в компонент Router (src/components/router/Router.js)

```
//...
cleanPath(path) {
  return path.replace(/\/\//g, '/');
}
//...
```

cleanPath использует метод String.replace для удаления любых двойных слешей из маршрута

Теперь, когда вы можете очистить вхождения слешей (//), следует проработать несколько других ситуаций для добавляемых вами маршрутов. Вы будете вызывать утилиту `normalizeRoute`, гарантирующую, что родительский и дочерний маршруты выполняются как правильные строки со слешами в нужных позициях (если они необходимы). Эта функция принимает маршрут и необязательный родительский элемент. Таким образом вы справитесь с несколькими ситуациями. В листинге 7.5 показано, как работает метод `normalizeRoute`.

Листинг 7.5. Создание утилиты normalizeRoute (src/components/router/Router.js)

```
//...
normalizeRoute(path, parent) {
  if (path[0] === '/') {
    return path;
  }

  if (parent == null) {
    return path;
  }

  return '${parent.route}/${path}';
}
//...
```

Функция получает маршрут и родительский объект, свойство route — это строка path

Если маршрут — это лишь символ /, можете просто вернуть его — нам не нужно присоединять его к предку

Если ни один из предков не предоставлен, можете просто вернуть path, потому что его не с чем объединять

Если есть предок, добавляйте путь к пути предка, объединяя их

7.2.4. Сопоставление URL-адресов и параметризованной маршрутизации

Вы создали вспомогательные инструменты, но еще не выполняете маршрутизацию. Чтобы начать сопоставление URL-адресов с компонентами, необходимо добавить маршруты в роутер. Как вы собираетесь делать это? По сути, вам нужно найти способ рендеринга данного компонента на основе того, что представляет собой текущий URL-адрес: «совпадающую» часть, о которой я постоянно говорю. Этот процесс не кажется трудоемким, но состоит из нескольких шагов.

Сначала рассмотрим ключевой компонент интерфейсной системы маршрутизации для браузера — *соответствие маршруту*. Нужно каким-то образом оценить строки маршрута и превратить их в значимые данные, пригодные для применения. Для этого вы будете задействовать небольшой пакет под названием `enroute`. Сам по себе он является крошечным роутером, который вы будете использовать для сопоставления маршрутов с вашими компонентами. `enroute` преобразует строки в регулярные выражения, которые можно применять для сопоставления строк (например, URL-адреса, по которым будет выполняться проверка). Вы также можете задействовать его для указания *параметров* маршрута, чтобы создать маршрут, например, `/users/:user` и получить доступ к идентификатору пользователя в `/users /1234` как что-то вроде `route.params.user` в своем коде. Ничего необычного, и вы, вероятно, видели что-то подобное, если работали с `express.js`.

Возможность настраивать URL-адреса полезна, потому что таким образом вы можете обрабатывать URL-адрес как другую форму ввода данных, которую можно передать роутеру. URL-адреса мощны отчасти благодаря динамичности. Они могут быть значимыми и позволяют пользователям обращаться к ресурсам напрямую, без предварительного посещения одной страницы и дальнейшего бестолкового серфинга, чтобы добраться туда, куда хочется.

Вы не будете использовать все возможности параметризации маршрутов, и мы рассмотрим лишь несколько примеров, чтобы убедиться, что вы понимаете, к чему стремитесь. В табл. 7.1 показаны URL-пути, которые могут пригодиться в обычном веб-приложении.

Таблица 7.1. Примеры общих маршрутов с параметрами

Маршрут	Пример использования
/	Домашняя страница приложения
/profile	Страница профиля пользователя; отображает настройки
/profile/settings	Настройки маршрута; дочерний элемент страницы профиля; отображает пользовательские настройки
/posts/:postID	postID, доступный для кода. Например, маршрут может выглядеть так: /posts/2391448. Полезен, если вы хотите создавать общедоступные ссылки на определенные сообщения
/users/:userID	:userID — параметр маршрута; полезен, чтобы отобразить конкретного пользователя на основе его идентификатора
/users/:userID/posts	Показать все сообщения пользователя; часть :userID URL-адреса динамична и доступна в вашем коде

Сейчас вы используете только одно свойство параметризованной маршрутизации с синтаксисом `:name`, но есть инструменты, которые позволят вам сделать гораздо больше. Если хотите подробнее узнать о параметризованной маршрутизации, познакомьтесь с библиотекой `path-to-regexp`, доступной по адресу www.npmjs.com/package/path-to-regexp. Это отличный инструмент — один из многих, на которые мы могли бы потратить время, но нам нужно сосредоточиться на текущей задаче — маршрутизации в React.

Важным выводом об инструментах маршрутизации `enroute` и `path-to-regexp` является то, что вы собираетесь применять их для упрощения сопоставления URL-адресов и работы с некоторыми параметрами маршрутов в URL-адресах. Сейчас не имеет значения, какой инструмент вы используете или хотите ли разработать собственный, — вам просто нужно что-то, что позволит сосредоточиться на фундаментальных принципах. Одна из прелестей React заключается в том, что вы сами можете принять обоснованное решение о том, какие инструменты маршрутизации хотите задействовать при создании приложений.

Упражнение 7.2. Размышления о параметрах

Параметризация маршрутов часто является полезным способом передачи данных в ваше приложение. Можете ли вы представить другие способы использования параметров маршрута, помимо получения идентификатора сообщения?

С помощью библиотеки сопоставления URL-адресов (`enroute`) вы станете определять, какой маршрут отобразить. Сейчас компонент `Router` имеет метод `render`, который ничего не делает, с него мы и начнем. В листинге 7.6 показано, как интегрировать `enroute` в роутер, и отображены итоговые изменения в методе `render`.

Листинг 7.6. Готовый роутер (src/components/router/Router.js)

```
import enroute from 'enroute';
import invariant from 'invariant';

export class Router extends Component {
  static propTypes = {
    children: PropTypes.element.isRequired,
    location: PropTypes.string.isRequired,
  }

  constructor(props) {
    super(props);

    // Хранение маршрутов в компоненте Router
    this.routes = {};

    // Настройка роутера для сопоставления и маршрутизации
    this.router = enroute(this.routes);
  }


  render() {
    const { location } = this.props;
    invariant(location, '<Router/> needs a location to work');
    return this.router(location);
  }
}
```

enroute — это крошечный функциональный роутер, который используется для сопоставления строковых URL-адресов и параметризации маршрутов

Установка propTypes как статического свойства класса

Настройка начального состояния компонента и инициализация enroute

Маршруты окажутся объектами с URL-адресами ключей

Передача маршрутов в enroute. Render будет задействовать возвращаемое enroute значение для установления соответствия URL-адресов компонентам

Передача текущего местоположения роутеру в качестве свойства

Использование invariant, чтобы убедиться в том, что вы не забыли указать местоположение

Наконец, и это самое главное, применение роутера для нахождения соответствия местоположению и возврата соответствующего компонента

Вы добавили немного кода, но некоторые из наиболее важных инструкций роутера уже на своем месте. На данном этапе нет никаких маршрутов для работы `enroute`, но основная механика реализована. Вам нужно найти компонент, связанный с маршрутом, а затем применить роутер для его отображения. В следующем подразделе вы создадите эти маршруты, чтобы роутер мог их использовать.

7.2.5. Добавление маршрутов в компонент Router

Чтобы добавить маршрут в роутер, вам понадобятся корректный строковый URL-адрес и компонент для него. Вы создадите метод компонента `Router`, который позволит вам связать эти два объекта: назовем его `addRoute`. Если вы взглянете на пример использования `enroute` на странице github.com/lapwinglabs/enroute, то увидите, как работает метод `enroute`. Он принимает объект со строковыми URL-адресами ключей и функциями для значений и, когда один из маршрутов сопоставлен, вызывает функцию и передает некоторые дополнительные данные. В листинге 7.7 показано, как вы задействовали бы библиотеку `enroute` без React. С помощью `enroute` можно сопоставлять функции, которые принимают параметры и любые дополнительные данные, и строки URL.

Листинг 7.7. Пример конфигурации маршрута (src/components/router/Router.js)

```
function edit_user (params, props) {
  return Object.assign({}, params, props)
}
```

Используются два параметра: параметры маршрута (например, /users /:user) и любые дополнительные данные, которые вы передаете

```
const router = enroute({
      '/users/new': create_user,
      '/users/:slug': find_user,
      '/users/:slug/edit': edit_user,
      '*': not_found
    });
```

Передача объекта с маршрутами и функциями, которые вы создали для обработки этих маршрутов

```
enroute('/users/mark/edit', { additional: 'props' })
```

Передача местоположения и любых дополнительных данных, чтобы выполнить правильную функцию

Теперь, когда вы получили представление о том, как работает `enroute` вне React, посмотрим, как интегрировать его в роутер и вдохнуть в него жизнь. Вместо того чтобы возвращать объект, как сделано в предыдущем листинге, вы вернете компонент. Но у вас пока нет способа добраться до маршрутов или их компонентов. Помните, как создавался компонент `Route`, который сохранил бы их, но ничего не отображал? Вам необходимо получить доступ к этим данным из своего родительского компонента (`Router`). Это означает, что нужно использовать свойство `children`.

ПРИМЕЧАНИЕ

Вы видели, как можно формировать компоненты в React для разработки новых компонентов путем создания родительско-дочерних отношений между ними. До сих пор вы использовали потомков извне, вкладывая компоненты друг в друга. Каждый раз, вкладывая и составляя компоненты, вы применяете концепцию React для потомков. Но вы еще не обращались динамически к любому из вложенных потомков из родительского компонента. Можете получить доступ к потомкам, которые передаются предкам как свойства компонента, как вы догадались, children.

Свойство `children`, доступное для каждого компонента или элемента React, — это то, что мы называем *непрозрачной* структурой данных, потому что оно, в отличие от почти всего остального в React, представляет собой не просто массив или объект старого доброго языка JavaScript. Возможно, ситуация изменится в будущих версиях React, но пока все это означает, что существует несколько инструментов, предоставляемых React, которые позволяют работать с `children`. Ряд методов доступен из `React.Children`, и для работы с непрозрачной структурой данных `children` вы можете использовать, в частности, следующие.

- `React.Children.map` — подобно `Array.map` в исходном JavaScript, он вызывает функцию для каждого ближайшего дочернего элемента внутри `children` (это означает, что он не будет проходить по всем возможным дочерним компонентам — только по прямым потомкам) и возвращает массив элементов, которые обходит. Возвращает `null` или `undefined`, а не пустой массив, если `children` имеет значение `null` или `undefined`:

  ```
  React.Children.map(children, function[(thisArg)])
  ```

- `React.Children.forEach` — работает так же, как `React.Chidlren.map`, но не возвращает массив:

  ```
  React.Children.forEach(children, function[(thisArg)])
  ```

- `React.Children.count` — возвращает общее количество компонентов, найденных в `children`. Оно равно количеству раз, когда `React.Children.map` или `React.Children.forEach` будет выполнять обратный вызов в одних и тех же элементах:

  ```
  React.Children.count(children)
  ```

- `React.Children.only` — возвращает единственного потомка из `children` или выдает ошибку:

  ```
  React.Children.only(children)
  ```

- `React.Children.toArray` — возвращает `children` как плоский массив с ключами, назначенными каждому потомку:

  ```
  React.Children.toArray(children)
  ```

Поскольку вы хотите добавить информацию маршрута к `this.routes` в компоненте `Router`, то станете использовать `React.Children.forEach` для итерации по каждому из дочерних элементов `Router` (помните, что это компоненты `Route`)

и получения доступа к их свойствам. С помощью этих свойств будете настраивать свои маршруты и указывать `enroute`, какой компонент следует отобразить для какого URL-адреса.

Самоуничтожающиеся компоненты в React

Версия React 16 позволила компонентам возвращать массивы из метода `render`. Ранее это было недоступно, а теперь открывает интересные возможности. Одна из них — идея *саморазрушающегося* или *самоуничтожающегося*[1] компонента. До этого, когда вы могли возвращать только один узел из любого компонента, вам часто приходилось обертывать компоненты в контейнер `div` или `span` только для получения правильного вывода JavaScript. Привычный сценарий выглядел бы примерно так:

```
export const Parent = () => {
  return (
    <Flex>
      <Sidebar/>
      <Main />
      <LinksCollection/>
    </Flex>
  );
}

export const LinksCollection = () => {
  return (
    <div>
      <User />
        <Group />
        <Org />
      </div>
  );
}
```

Компоненты верхнего уровня, один за другим выложенные с помощью Flexbox (или CSS grids)

Обертка div добавлена, потому что User, Group и Org не могут быть возвращены вместе в JavaScript — язык не поддерживает несколько возвращаемых значений

Это сильно раздражало многих разработчиков, хотя, конечно же, не помешало использовать React. Однако одна из основных проблем заключается не только в том, что обертка `div` оказывается ненужной. Как вы здесь видите, приложение подготовлено с помощью библиотеки Flexbox (или другого API CSS, который не сработал бы в этом сценарии).

Проблема, создаваемая оберткой `div`, заключается в том, что она заставляет перемещать компоненты на уровень выше, чтобы они не были сгруппированы в одном узле. Есть, конечно, и другие причины, вызывающие проблемы и поиск обходных путей, но с этой я сталкивался много раз.

С появлением React 16 и последующих версий стало возможно возвращать массивы, так что теперь есть способ решить и эту задачу. React 16 предоставила множество других

[1] Большое спасибо Бену Илегбоду за то, что рассказал мне об этой идее!

мощных функций, но массивы были самым приятным изменением. Теперь разработчики могут сделать что-то подобное:

```
export const SelfEradicating = (props) => props.children
```

Этот компонент действует как своего рода сквозной проход, уходящий с маршрута или самоуничтожающийся по мере отображения своих потомков. Используя этот подход, вы способны поддерживать разделение компонентов без необходимости подстраховки в таких случаях, как техника компоновки CSS. Тот же сценарий с самоуничтожающимся компонентом может выглядеть примерно так:

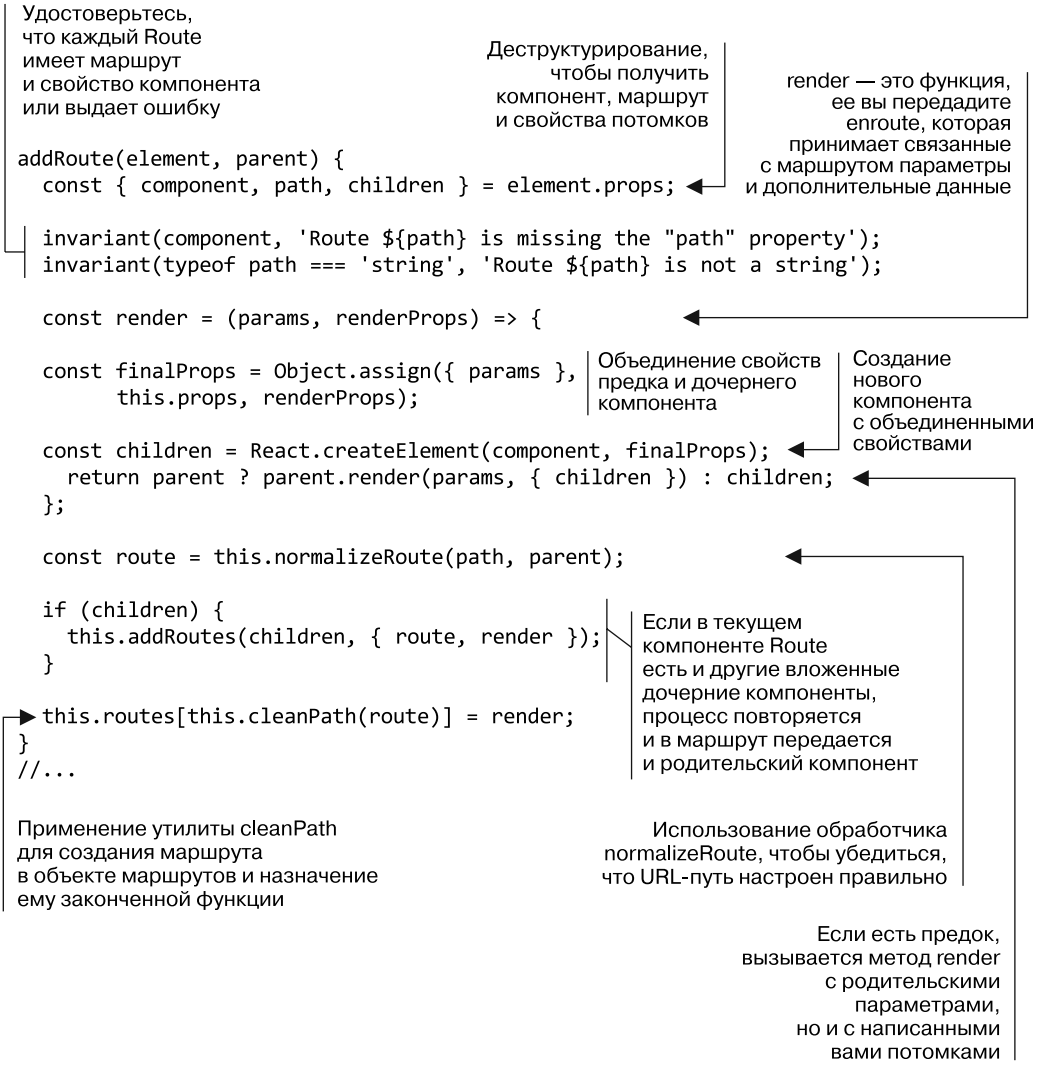
```
export const SelfEradicating = (props) => props.children

export const Parent = () => {
  return (
    <Flex>
      <Sidebar/>
      <Main />
      <LinksCollection/>
    </Flex>
  );
}

export const LinksCollection = () => {
  return (
    <SelfEradicating>
      <User />
      <Group />
      <Org />
    </SelfEradicating>
  );
}
```

Помните, что метод `enroute` ожидает, что вы дадите функцию каждому маршруту, чтобы он мог передать в нее информацию о параметрах и другие данные? Эта функция — место, где вы указываете React создать компонент и поддержать рендеринг дополнительных дочерних компонентов. В листинге 7.8 показано, как добавить компоненту методы `addRoute` и `addRoutes`. Метод `addRoutes` задействует `React.Children.forEach` для итерации по дочерним компонентам `Route`, захвата их данных и настройки маршрута для использования в `enroute`. Это основная часть роутера — как только вы ее реализуете, он будет готов к запуску!

Упражнение 7.3. props.children

В этой главе мы говорили о `props.children` React. Существуют ли различия между `props.children` и другими свойствами? Почему могут существовать какие-то различия?

Листинг 7.8. Методы addRoute и addRoutes (src/components/router/Router.js)




```
addRoute(element, parent) {
  const { component, path, children } = element.props;

  invariant(component, 'Route ${path} is missing the "path" property');
  invariant(typeof path === 'string', 'Route ${path} is not a string');

  const render = (params, renderProps) => {

  const finalProps = Object.assign({ params },
        this.props, renderProps);

  const children = React.createElement(component, finalProps);
    return parent ? parent.render(params, { children }) : children;
  };

  const route = this.normalizeRoute(path, parent);

  if (children) {
    this.addRoutes(children, { route, render });
  }

  this.routes[this.cleanPath(route)] = render;
}
//...
```

Уф! В этих нескольких строках кода происходило многое. Не стесняйтесь возвратиться к нему пару раз, чтобы убедиться, что поняли все концепции. Когда вы добавите метод `addRoutes`, мы рассмотрим шаги и пересмотрим рендеринг. Но сначала вы добавите метод `addRoutes`. Это довольно просто. В листинге 7.9 показано, как это сделать.

Теперь роутер закончен и готов к работе (рис. 7.3). В листинге 7.10 показан компонент `Router` в итоговом состоянии со вспомогательными инструментами (нормализация маршрута, инвариантное применение), которые для краткости опущены. В следующей главе вы введете компонент `Router` в действие.

Листинг 7.9. Метод addRoutes (/components/router/Router.js)

```
//...
constructor(props) {
  super(props);
  this.routes = {};
  this.addRoutes(props.children);
  this.router = enroute(this.routes);
}

addRoutes(routes, parent) {
  React.Children.forEach(routes, route => this.addRoute(route, parent));
}
```

Несмотря на то что addRoutes используется в методе addRoute, добавление его в конструктор компонента позволяет начать настройку маршрутов

Метод addRoutes применяется в addRoute всегда, когда есть дополнительные потомки для итерации

Использование утилиты React.Children.forEach для итерации по каждому из дочерних элементов, а затем вызов addRoute для каждого дочернего компонента Route

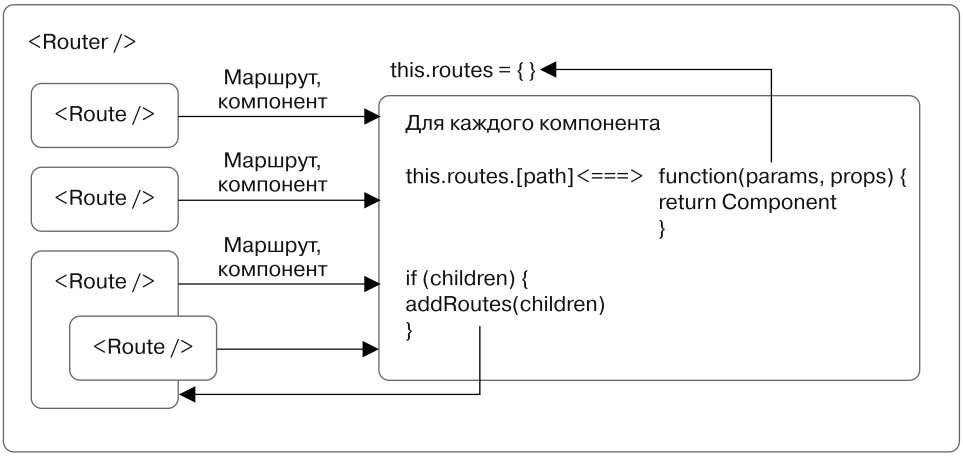


Рис. 7.3. Процесс добавления маршрутов в роутер. Для каждого компонента Route, найденного в компоненте Router, извлеките маршрут и свойства компонента, а затем задействуйте их для создания функции, которую вы можете связать с URL-адресом для использования в enroute. Если у Route есть дочерние компоненты, запустите тот же процесс и для них, прежде чем читать дальше. Когда все будет готово, свойство routes получит все необходимые маршруты

Листинг 7.10. Законченный Router (src/components/router/Router.js)

```
import PropTypes from 'prop-types';

import React, { Component } from 'react';
import enroute from 'enroute';
import invariant from 'invariant';

export default class Router extends Component {
  static propTypes = {
```

```
    children: PropTypes.array,
    location: PropTypes.string.isRequired
  };

  constructor(props) {
    super(props);

    this.routes = {};

    this.addRoutes(props.children);
    this.router = enroute(this.routes);
  }

  addRoute(element, parent) {
    const { component, path, children } = element.props;

    invariant(component, 'Route ${path} is missing the "path" property');
    invariant(typeof path === 'string', 'Route ${path} is not a string');

    const render = (params, renderProps) => {
      const finalProps = Object.assign({ params }, this.props, renderProps);

      const children = React.createElement(component, finalProps);

      return parent ? parent.render(params, { children }) : children;
    };

    const route = this.normalizeRoute(path, parent);

    if (children) {
      this.addRoutes(children, { route, render });
    }

    this.routes[this.cleanPath(route)] = render;
  }

  addRoutes(routes, parent) {
    React.Children.forEach(routes, route => this.addRoute(route, parent));
  }

  cleanPath(path) {
    return path.replace(/\/\//g, '/');
  }

  normalizeRoute(path, parent) {
    if (path[0] === '/') {
      return path;
    }
    if (!parent) {
      return path;
    }
```

```
    return '${parent.route}/${path}';
  }

  render() {
    const { location } = this.props;
    invariant(location, '<Router/> needs a location to work');
    return this.router(location);
  }
}
```

7.3. Резюме

В этой главе вы начали превращать React-приложение из простой страницы с компонентами в более надежное приложение, которое поддерживает маршрутизацию и настройку маршрутов. Мы рассмотрели довольно много вопросов и изучили расширенное использование компонентов для создания роутера с нуля.

- Маршрутизация в современных клиентских приложениях не требует перезагрузки всей страницы. Вместо этого ее можно обработать с помощью клиентских приложений, таких как React. Это также может уменьшить время загрузки браузера и потенциально нагрузку на сервер.
- React не имеет встроенной библиотеки маршрутизации, реализованной в некоторых фреймворках. Вместо этого можно либо выбрать одну из библиотек, либо создать собственный роутер с нуля (как вы и сделали).
- React предоставляет несколько утилит для работы с непрозрачной структурой данных `children`. Вы можете итерировать множество компонентов, проверять их количество и делать многое другое.
- Используйте выполненную вами настройку маршрутизации для динамического изменения того, какие дочерние элементы отображаются внутри компонента. Вы отслеживаете изменения в местоположении и рендерите контент в браузере, учитывая полученные данные.

В следующей главе вы запустите роутер и добавите аутентификацию в свое приложение с помощью Firebase.

8 Маршрутизация и интеграция Firebase

- Использование роутера, разработанного в главе 7.
- Создание связанных с маршрутизацией компонентов, таких как Router, Route и Link.
- Работа с API HTML5 History для включения маршрутизации с обновлением состояния.
- Многократное использование компонентов.
- Интеграция аутентификации пользователей и Firebase.

В предыдущей главе вы с нуля собрали простой роутер, чтобы разобраться, как выполнять маршрутизацию в React-приложениях. Здесь же возьмете созданный роутер и разделите приложение Letters Social для более эффективной работы. В конце главы вы сможете перемещаться по своему приложению, просматривать отдельные страницы сообщений и выполнять аутентификацию пользователей.

Получение исходного кода

Как и прежде, вы можете получить исходный код примеров из этой главы, перейдя в репозиторий GitHub по адресу github.com/react-in-action/letters-social. Если планируете начать работу здесь самостоятельно с нуля, возьмите исходный код примеров из глав 5 и 6 (если изучили их и сами выполнили примеры) или обратитесь к ветви, относящейся к данной главе (chapter-7-8).

Помните, что каждая ветвь содержит итоговый код главы (например, chapter-7-8 содержит код, получаемый в конце глав 7 и 8). Вы можете выполнить в оболочке командной строки одну из следующих команд по своему выбору, чтобы получить код примеров из текущей главы. Если репозитория вообще нет, выполните такую команду:

```
git clone git@github.com:react-in-action/letters-social.git
```

Если у вас уже клонирован репозиторий, то следующую:

```
git checkout chapter-7-8
```

Возможно, вы перешли сюда из другой главы, поэтому стоит проверить, установлены ли у вас все нужные зависимости, с помощью следующей команды:

```
npm install
```

8.1. Использование роутера

В предыдущей главе с помощью React вы создали работающий роутер. Занимаясь приложением React в рабочей среде, вы, вероятно, выберете что-то вроде `React Router`. К счастью, `React Router` применяет очень похожий API, но имеет более продвинутые функции, которые позволяют достичь новых высот маршрутизации. Возможно, вам не нужны все эти особенности и разработанного проекта достаточно. Это прекрасно — выберите инструменты, лучше всего подходящие для решения проблем, а не те, что популярны на сайтах типа GitHub или Hacker News. Ваши потребности изменятся, когда в главе 12 мы займемся проблемами рендеринга на стороне сервера, а здесь освоим `React Router`.

Начнем с нового роутера. Сначала нужно подключить его к API HTML5 History (developer.mozilla.org/en-US/docs/Web/API/History), чтобы пользоваться навигацией, не требующей полной перезагрузки страницы. Задействуйте навигацию *с изменением состояния*, потому что не нужно каждый раз запрашивать сервер для обновления полной страницы. Можете также использовать маршрутизацию на основе хеша (подробнее см. github.com/ReactTraining/react-router/blob/v3/docs/guides/Histories.md).

Мы не станем тратить много времени на изучение API HTML5, потому что они заслуживают отдельной книги. Вы будете применять популярную библиотеку `history`, доступную по адресу www.npmjs.com/package/history. Она позволит работать с API History в надежном и предсказуемом виде в браузерах. Чтобы убедиться, что она установлена, выполните команду `npm install --save history`. После установки библиотеки нужно внести некоторые изменения в файл `index.js` — на данный момент корневой объект для всего приложения. До сих пор этот файл использовался для того, чтобы React DOM превращал ваше приложение в DOM-элемент. Но у вас включена маршрутизация, и компонент `Router` ожидает местоположение (см. главу 7). Вам нужно найти способ передать это местоположение и воспользоваться преимуществами API History HTML5, задействуя библиотеку `history`, а файл `index.js` — идеальное место для этого.

Упражнение 8.1. Сравнение клиентской и серверной маршрутизации

Найдите минутку, чтобы рассмотреть разницу между маршрутизацией на стороне клиента и клиент-серверной маршрутизацией на основе URL-адреса. Каковы основные различия между маршрутизацией на стороне клиента и маршрутизацией на стороне сервера?

Вдобавок к использованию преимуществ `history` вам необходимо настроить маршруты. Для этого следует реорганизовать некоторые из компонентов, чтобы получить представление о преимуществах сочетаемости и модульности в React. Вы займетесь перестановками, но не должны кардинально изменять способ работы своих компонентов. Для начала посмотрим, как изменить компонент `App`. Он должен

служить контейнером для дочерних маршрутов, потому что требуется, чтобы на каждой странице были одинаковые боковые панели и панель навигации, а изменялся только контент свойства `children`. На рис. 8.1 показан пример результата.

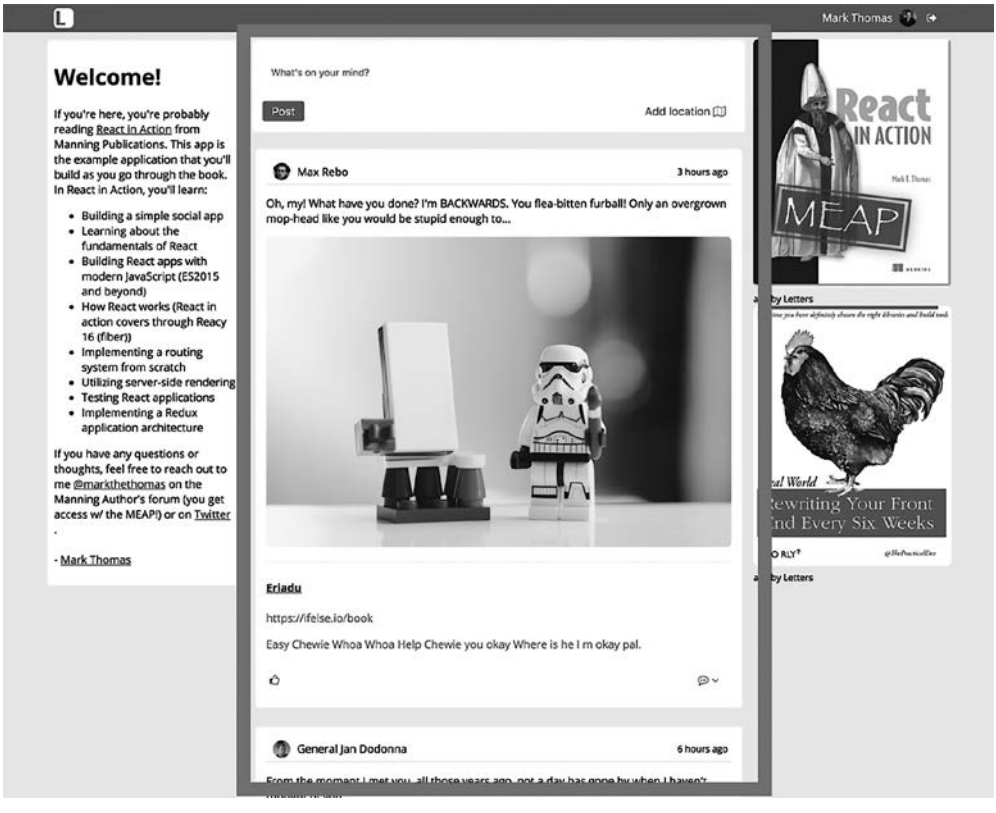


Рис. 8.1. Выделенная область меняется в зависимости от того, какой вид вы решите отображать на основе URL-адреса. Со временем можете даже использовать большую вложенность и расширить эту область, включив в нее боковые панели, чтобы поддерживать одну и ту же навигационную панель на разных страницах и иметь другие маршруты, в которых есть динамические области

Чтобы добиться такого рода вложенности, необходимо реорганизовать компонент `App`, чтобы динамически показывать `children`, как показано в листинге 8.1. К счастью, вы не удалите большую часть написанного кода, а просто переместите его. По мере рефакторинга вы реорганизуете файлы своего приложения. Создайте в папке `src` новый каталог с именем `pages`. Размещайте в нем компоненты, которые, как правило, содержат только другие компоненты и предоставляют им данные. Я расскажу об этом подробнее, когда мы будем исследовать архитектуру React-приложения в последующих главах.

Листинг 8.1. Рефакторинг компонента App (src/app.js)

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

import ErrorMessage from './components/error/Error';
import Nav from './components/nav/navbar';
import Loader from './components/Loader';

class App extends Component {
  constructor(props) {
    super(props);
    this.state = {
      error: null,
      loading: false
    };
  }
  static propTypes = {
    children: PropTypes.node
  };
  componentDidCatch(err, info) {
    console.error(err);
    console.error(info);
    this.setState(() => ({
      error: err
    }));
  }
  render() {
    if (this.state.error) {
      return (
        <div className="app">
          <ErrorMessage error={this.state.error} />
        </div>
      );
    }
    return (
      <div className="app">
        <Nav user={this.props.user} />
        {this.state.loading ? (
          <div className="loading">
            <Loader />
          </div>
        ) : (
          this.props.children
        )}
      </div>
    );
  }
}

export default App;
```

Настройка границы ошибки верхнего уровня с помощью компонента componentDidCatch, чтобы выводилась ошибка, если что-то пошло не так

Отображение ошибки, если она возникла

Передача свойств пользователя — применяется при интеграции с Firebase

Если приложение находится в состоянии загрузки, рендерится загрузчик

Использование props.children для вывода текущего активного маршрута

Вам нужно создать компонент для главной страницы, чтобы пользователи видели сообщения. Создайте файл `home.js` и поместите его в каталог `pages`. Он должен быть вам знаком — это основной компонент, который уже существовал, прежде чем вы разделили контент на страницы. В листинге 8.2 показан код компонента `Home` с логикой метода, который вы реализовали, прежде чем закомментировать для краткости. Помните, что, как и во всех прочих главах, можете сверить его с кодом из других глав, если хотите увидеть, как изменилось приложение, или посмотреть итоговую версию на странице github.com/react-in-action/letters-social.

Листинг 8.2. Перестроенный компонент Home (src/pages/Home.js)

```
import React, { Component } from 'react';
import parseLinkHeader from 'parse-link-header';
import orderBy from 'lodash/orderBy';

import * as API from '../shared/http';
import Ad from '../components/ad/Ad';
import CreatePost from '../components/post/Create';
import Post from '../components/post/Post';
import Welcome from '../components/welcome/Welcome';

export class Home extends Component {
  constructor(props) {
    super(props);
    this.state = {
      posts: [],
      error: null,
      endpoint: '${process.env

  .ENDPOINT}/posts?_page=1&_sort=date&_order=
  DESC&_embed=comments&_expand=user&_embed=likes'
    };
    this.getPosts = this.getPosts.bind(this);
    this.createNewPost = this.createNewPost.bind(this);
  }
  componentDidMount() {
    this.getPosts();
  }
  getPosts() {
    API.fetchPosts(this.state.endpoint)
      .then(res => {
        return res.json().then(posts => {
          const links = parseLinkHeader(res.headers.get('Link'));
          this.setState(() => ({
            posts: orderBy(this.state.posts.concat(posts), 'date', 'desc'),
            endpoint: links.next.url,
          }));
        });
      })
      .catch(err => {
        this.setState(() => ({ error: err }));
      });
  }
  createNewPost(post) {
```

Не забывайте менять пути импорта — компонент находится в другом каталоге

Логика для них точно такая же: вы только перемещаете компоненты, чтобы создать новую иерархию

```
    post.userId = this.props.user.id;
    return API.createPost(post)
      .then(res => res.json())
      .then(newPost => {
        this.setState(prevState => {
          return {
            posts: orderBy(prevState.posts.concat(newPost),
            'date', 'desc')
          };
        });
      })
      .catch(err => {
        this.setState(() => ({ error: err }));
      });
  }
  render() {
    return (
      <div className="home">
        <Welcome />
        <div>
          <CreatePost onSubmit={this.createNewPost} />
          {this.state.posts.length && (
            <div className="posts">
              {this.state.posts.map(({ id }) => {
                return <Post id={id} key={id}
                user={this.props.user} />;
              })}
            </div>
          )}
          <button className="block" onClick={this.getPosts}>
            Load more posts
          </button>
        </div>
        <div>
          <Ad url="https://ifelse.io/book"
          imageUrl="/static/assets/ads/ria.png" />
          <Ad url="https://ifelse.io/book"
          imageUrl="/static/assets/ads/orly.jpg" />
        </div>
      </div>
    );
  }
}

export default Home;
```

Логика для них точно такая же: вы только перемещаете компоненты, чтобы создать новую иерархию

Теперь, когда компонент `Home` перемещен, вы готовы настроить маршруты и применить инструмент `history`, чтобы роутер мог реагировать на изменения местоположения браузера. Часто бывает полезно сделать один модуль доступным для других частей приложения в качестве утилиты, чтобы избежать лишней работы. Мы продолжим в том же духе и в дальнейшем, но вы, вероятно, уже сделали это. Вы добьетесь этого с помощью библиотеки `history`, как показано в листинге 8.3, потому что станете использовать ее, кроме прочего, и для создания ссылок, поддерживаемых роутером, которые не должны быть обычными тегами типа `<a href = " "> </>`.

Листинг 8.3. Настройка библиотеки history (src/history/history.js)

```
import createHistory from 'history/createBrowserHistory';
const history = createHistory();
const navigate = to => history.push(to);
export { history, navigate };
```

Назначение одного экземпляра библиотеки history доступным приложению

Экспорт метода навигации и экземпляра истории (на случай, если позже понадобится прямой доступ)

Теперь, когда `history` настроена, можете настроить остальную часть файла `index.js` и сконфигурировать роутер. В листинге 8.4 показано, как это сделать.

Листинг 8.4. Настройка файла index.js для маршрутизации (src/index.js)

```
import React from 'react';
import { render } from 'react-dom';

import { App } from './pages/App';
import { Home } from './pages/Home';
import Router from './components/router/Router';
import Route from './components/router/Route';
import { history } from './history';

import './shared/crash';
import './shared/service-worker';
import './shared/vendor';
import './styles/styles.scss';

export const renderApp =
  (state, callback = () => {}) => {
  render(
    <Router {...state}>
      <Route path="" component={App}>
        <Route path="/" component={Home} />
      </Route>
    </Router>,
    document.getElementById('app'),
    callback
  );
};

let state = {
  location: window.location.pathname,
};

history.listen(location => {
  state = Object.assign({}, state, {
    location: location.pathname
  });
  renderApp(state);
});

renderApp(state);
```

Импорт React DOM

Импорт компонентов App, Home, Router и Route

Импорт утилиты history, которую вы только что написали

Создание функции, которую будете вызывать для рендеринга приложения. Оборачивание метода render React DOM, чтобы стало возможно передавать данные о местоположении и обратный вызов

Создание маршрута для компонентов App и Home

Использование оператора распространения JSX для заполнения местоположения в качестве свойств для компонента Router

Рендеринг приложения, чтобы нацелить DOM-элемент в index.html

Создание объекта состояния для отслеживания местоположения и пользователя

Сигнализация при изменении местоположения и обновление роутера, что приведет к повторному рендерингу приложения с новыми данными состояния

Рендеринг приложения

8.1.1. Создание страницы для сообщения

Вы маршрутизируете! Теперь сделано достаточно, чтобы настроить маршрутизацию в приложении. Но вы не сделали ничего, чтобы пользователь мог перемещаться по различным частям приложения. Сейчас в приложении, вероятно, станет больше страниц и подразделов. Если бы разрабатывалась более сложная версия приложения для социальных сетей, у вас, вероятно, существовали бы разделы для страницы профиля, пользовательских настроек, сообщений и т. д. Но в данном случае требуется всего лишь отображать отдельные сообщения. Как это сделать? Начнем с URL-адреса. Помните, что маршрут `/posts/:postID` уже несколько раз использовался в примерах? Ваши страницы публикаций будут доступны по этому URL-адресу.

Начнем с разработки компонента страницы для отдельных сообщений. В предыдущих главах разработан компонент Post, который извлекает свои данные сразу после загрузки, поэтому создание такой страницы с одним сообщением не должно быть слишком сложным. Вам нужно выполнить новый компонент для этой страницы, убедиться, что сообщение включено и вы правильно сопоставляете его с маршрутом. Иным будет лишь место, откуда вы получаете идентификатор сообщения. Вместо первоначальной выборки с сервера он будет доставлен из URL-адреса. Вы использовали специальный синтаксис для настройки URL-адреса, и роутер обеспечит доступность параметризованных данных маршрута для вашего компонента. В листинге 8.5 показано, как настроить страницу с одним сообщением.

Листинг 8.5. Создание компонента SinglePost (src/pages/Post.js)

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

import Ad from '../components/ad/Ad';
import Post from '../components/post/Post';

export class SinglePost extends Component {
  static propTypes = {
    params: PropTypes.shape({
      postId: PropTypes.string.isRequired
    })
  };
  render() {
    return (
      <div className="single-post">
        <Post id={this.props.params.postId} />
        <Ad
          url="https://www.manning.com/books/react-in-action"
          imageUrl="/static/assets/ads/ria.png"
        />
      </div>
    );
  }
}

export default SinglePost;
```

Импортирование компонента Post, созданного в предыдущих главах

Получение идентификатора сообщения из свойств, переданных роутером

Теперь, когда подходящий компонент найден, можете интегрировать его в свой роутер, чтобы пользователи могли перемещаться по индивидуальным сообщениям. В листинге 8.6 показано, как добавить компонент `SinglePost` в роутер. Обратите внимание на то, что вы пользуетесь преимуществами *параметризованной* маршрутизации, которую видели в предыдущих примерах роутеров. Часть маршрута `:post` предоставляется вашему компоненту в свойстве `params`.

Листинг 8.6. Добавление индивидуальных сообщений в роутер (src/index.js)

```
import React from 'react';
import { render } from 'react-dom';

import * as API from './shared/http';
import { history } from './history';
import Route from './components/router/Route';
import Router from './components/router/Router';
import App from './app';
import Home from './pages/home';
import SinglePost from './pages/post';

//...

export const renderApp = (state, callback = () => {}) => {
  render(
    <Router {...state}>
      <Route path="" component={App}>
        <Route path="/" component={Home} />
        <Route path="/posts/:postId" component={SinglePost} />
      </Route>
    </Router>,
    document.getElementById('app'),
    callback
  );
};

//...
```

Импортирование компонента SinglePost для использования в роутере

Настройка маршрута SinglePost с применением специального параметризованного синтаксиса маршрутизации (:post)

8.1.2. Создание компонента <Link/>

Если вы запустите свое приложение в режиме разработки и попробуете пощелкать кнопкой мыши, то заметите, что, хотя у вас все еще есть маршруты, настроенные для отдельных сообщений, вы не можете перейти туда, не зная идентификатора сообщения, так как не можете поместить его в URL-адрес. Это досадно, не так ли?

Вам нужно сделать пользовательский компонент `Link`, который будет работать с инструментом `history` и роутером. В противном случае пользователи, вероятно, быстро закроют ваше приложение, а инвесторы будут недовольны. Как можно решить эту задачу? Обычный элемент привязки (`<a href="/">Link!</a>`) не подходит, потому что он попытается перезагрузить всю страницу, а это нам не нужно. Возможно, вы также захотите создать ссылки на объекты, которые не являются

элементами привязки, например сообщение в списке или все, что вы не хотите оборачивать в элемент привязки.

ПРИМЕЧАНИЕ

Доступность — это степень, в которой интерфейс может использоваться кем-то. Вероятно, вы уже слышали о веб-доступности, возможно, даже что-то знаете о ней. Все в порядке — легко научиться обеспечивать веб-доступность. Вам нужно убедиться, что приложением удобно пользоваться как можно большему числу людей независимо от того, делают ли они это с помощью мыши и клавиатуры, устройства для чтения с экрана или других гаджетов. Я только что упомянул о создании произвольных элементов приложения, по которым можно перемещаться с помощью компонента Link, что должно быть сделано осторожно с точки зрения доступности. Учитывая это, я хотел бы здесь лишь кратко упомянуть о доступности. Поскольку разработка доступных веб-приложений — огромная и важная тема, она выходит за рамки данной книги. Существуют компании, приложения и хобби-проекты, которые считают ее достижение первоочередной задачей. Хотя вы можете рассматривать исходный код для Letters Social как способ создания приложений с помощью компонентов React, мы не занимаемся всеми проблемами доступности, которые могут возникнуть в вашем приложении. Чтобы узнать больше о доступности, ознакомьтесь в Интернете с авторскими практиками WAI-ARIA (www.w3.org/WAI/PF/aria-practices) или документацией MDN по ARIA (developer.mozilla.org/en-US/docs/Web/Accessibility/ARIA). Ари Риццитано также собрал отличный материал по этой теме, уделяя особое внимание доступности в React. Он называется «Создание доступных компонентов» (speakerdeck.com/arizzitano/building-accessible-components).

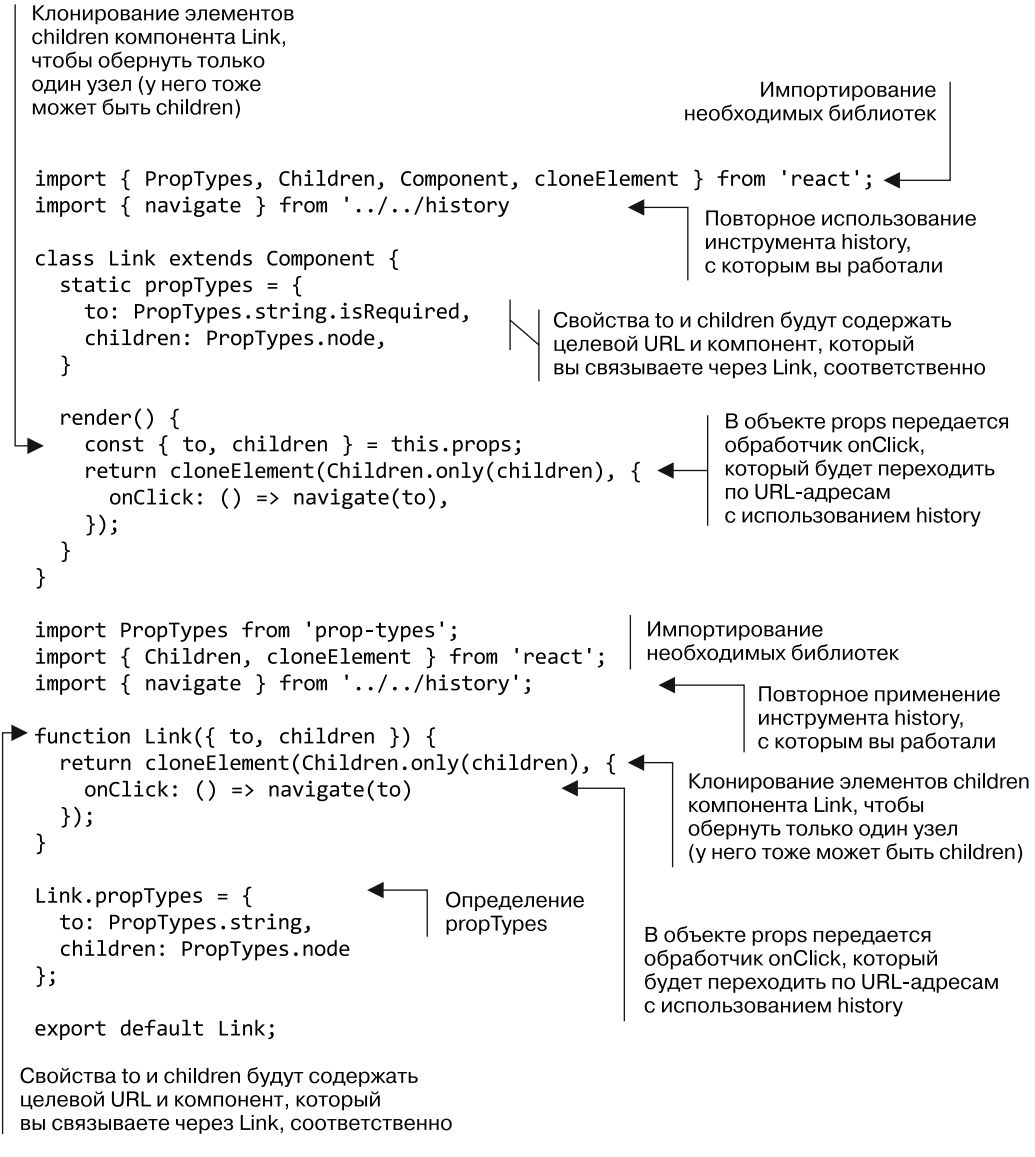
Вам снова понадобится применять инструмент `history` и интегрировать его в компонент `Link`, который можно использовать для включения связывания методом обновления состояния в истории в вашем приложении. Помните функцию `navigate`, которую вы открыли ранее? Работая с ней, вы можете программно указать библиотеке `history`, что нужно изменить местоположение пользователя. Чтобы превратить эту функциональность в компонент, используйте некоторые утилиты React для переноса других компонентов в интерактивный компонент `Link`. Вы возьмете `React.cloneElement` для создания копии целевого элемента, а затем добавите обработчик щелчков кнопкой мыши, который будет выполнять навигацию. Сигнатура `React.cloneElement` выглядит примерно так:

```
ReactElement cloneElement(
    ReactElement element,
     [object props],
     [children ...]
)
```

Утилита принимает элемент для клонирования, `props` для слияния с новым элементом и любые `children`, которыми он должен обладать. Вы будете применять эту утилиту для клонирования компонента, который хотите превратить в `Link`. Нужно убедиться, что компонент `Link` имеет только один дочерний элемент,

поэтому используйте инструмент `React.Children.only`, упомянутый ранее в этой главе. Все вместе эти инструменты позволят вам превратить другие компоненты в компоненты `Link`, что поможет пользователю перемещаться по приложению. В листинге 8.7 показано, как выполнить компонент `Link`.

Листинг 8.7. Создание компонента Link (src/components/router/Link.js)



Чтобы интегрировать компонент `Link`, можете обернуть отдельные сообщения в компонент многократного использования `Post` и убедиться, что `Link` получает

свойство `to`, которое отправит пользователя на нужную страницу (см. предыдущую заметку о доступности). Придерживайтесь этого шаблона, чтобы обернуть другие компоненты аналогичным образом и превратить их в компоненты `Link`. В листинге 8.8 показано, как интегрировать компонент `Link`.

Листинг 8.8. Интеграция компонента Link (src/components/post/Post)

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

import * as API from '../../shared/http';
import Content from './Content';
import Image from './Image';
import Link from './Link';
import PostActionSection from './PostActionSection';
import Comments from '../comment/Comments';
import DisplayMap from '../map/DisplayMap';
import UserHeader from '../post/UserHeader';

import RouterLink from '../router/Link';

export class Post extends Component {

//...

  render() {
    return this.state.post ? (
      <div className="post">
        <RouterLink to={'/posts/${this.state.post.id}'}>
          <span>
            <UserHeader date={this.state.post.date}
            user={this.state.post.user} />
            <Content post={this.state.post} />
            <Image post={this.state.post} />
            <Link link={this.state.post.link} />
          </span>
        </RouterLink>
        {this.state.post.location && <DisplayMap
        location={this.state.post.location} />}
        <PostActionSection showComments={this.state.showComments} />
        <Comments
          comments={this.state.comments}
          show={this.state.showComments}
          post={this.state.post}
          handleSubmit={this.createComment}
          user={this.props.user}
        />
      </div>
    ) : null;
  }
}

export default Post;
```

Импортирование компонента Link и присвоение ему псевдонима RouterLink, чтобы избежать конфликта имен с компонентом Link, используемым в сообщениях

Обертывание раздела компонента Post, который требуется связывать, и присвоение ему корректного идентификатора

Теперь роутер полностью интегрирован в приложение. С этого момента пользователи могут просматривать отдельные сообщения, что отлично подходит для совместного применения и фокусировки на одном сообщении в определенный момент. Инвесторы будут приятно удивлены. Но вы еще не закончили. В следующем подразделе обсудим, что делать, если вы не можете сопоставить URL-адрес с компонентом.

Упражнение 8.2. Добавление ссылок

Попробуйте найти другие области приложения, в которых можно использовать компонент `Link`, чтобы превратить их в ссылки. *Подсказка*: как пользователи смогут вернуться на домашнюю страницу после перехода на отдельное сообщение? Вдобавок подумайте о впечатлении, которое складывается у пользователя при перемещении по приложению. Что для него было бы целесообразным? Какие из областей приложения вы превратили бы в `Link`? Случалось вам превращать что-то в `Link`, хотя оно еще не было элементом привязки? Просмотрите одну страницу сообщения в исходном коде приложения, чтобы увидеть пример добавления простой кнопки Назад.

8.1.3. Создание компонента <NotFound/>

Перейдите к каталогу `/oops` в приложении Letters и посмотрите, что случится. Ничего? Да, это то, что должно произойти в результате выполнения вашего кода, но не то, что хотелось бы показать пользователям. Прямо сейчас ваш компонент `Router` не обрабатывает маршруты «не найдено» или «поймать все». Вы хотите быть вежливыми со своими пользователями и предполагаете, что в какой-то момент они (или вы) могут совершить ошибку и попытаться перейти к маршруту, которого нет в приложении. Чтобы решить эту проблему, создайте простой компонент `NotFound` и настройте его при формировании экземпляра компонента `Router`. В листинге 8.9 показано, как реализовать компонент `NotFound`.

Листинг 8.9. Создание компонента NotFound (src/pages/404.js)

```
import React from 'react';
import Link from '../components/router/Link';

export const NotFound = () => {
  return (
    <div className="not-found">
      <h2>Not found :(</h2>
      <Link to="/">
        <button>go back home</button>
      </Link>
    </div>
  );
};

export default NotFound;
```

Импортирование компонента Link, чтобы пользователи могли вернуться на домашнюю страницу

Нет необходимости в состоянии компонента, поэтому создается функциональный компонент без состояния

Использование компонента Link, чтобы пользователи могли вернуться на домашнюю страницу

Теперь, когда компонент `NotFound` существует, необходимо интегрировать его в конфигурацию роутера. Возможно, вам интересно, как сообщить роутеру, что он должен отправлять пользователей к компоненту `NotFound`. Ответ таков: используйте символ * при настройке роутера. Он обозначает любое совпадение, и если вы поместите его в конец конфигурации, туда будут вести все маршруты, которые не сопоставлены ранее. Не забудьте, что порядок имеет значение: если вы поместите «все маршруты» слишком высоко, пункт будет соответствовать чему угодно и не станет работать так, как вам бы хотелось. В листинге 8.10 показано, как настроить дополнительные маршруты в роутере.

Листинг 8.10. Добавление отдельных сообщений в роутер (src/index.js)

```
//...
import NotFound from './pages/404 ';
//...

export const renderApp = (state, callback = () => {}) => {
  render(
    <Router {...state}>
      <Route path="" component={App}>
        <Route path="/" component={Home} />
        <Route path="/posts/:postId" component={SinglePost} />
        <Route path="*" component={NotFound} />
      </Route>
    </Router>,
    document.getElementById('app'),
    callback
  );
};
//...
```

Импортирование компонента NotFound

Настройка маршрута для компонента NotFound, чтобы он служил маршрутом для всех непрописанных адресов

8.2. Интеграция Firebase

Теперь, когда роутер полностью настроен и функционирует, рассмотрим еще одну область, чего мы не могли сделать раньше: возможность авторизации и аутентификации пользователя. Примените для этого популярную и удобную платформу Firebase «серверная часть как служба» (firebase.google.com). Firebase — это службы, которые абстрагируют или заменяют интерфейс серверного API, обрабатывающий пользовательские данные, аутентификацию и другие опции. Нам требуется замена серверного API.

Вы задействуете Firebase не для полной замены серверной части своего приложения (вы все еще используете собственный сервер API), а для допуска авторизации и управления пользователями. Чтобы начать работу с Firebase, перейдите на страницу firebase.google.com и создайте учетную запись, если до сих пор не сделали этого. После регистрации перейдите в консоль Firebase на странице console.firebase.google.com и разработайте проект, который будет использоваться в Letters Social. После этого нажмите кнопку Add Firebase to Your Web App (Добавить Firebase в веб-приложение), чтобы открыть модальное наложение. Вы увидите некоторые сведения о конфигурации приложения (рис. 8.2).

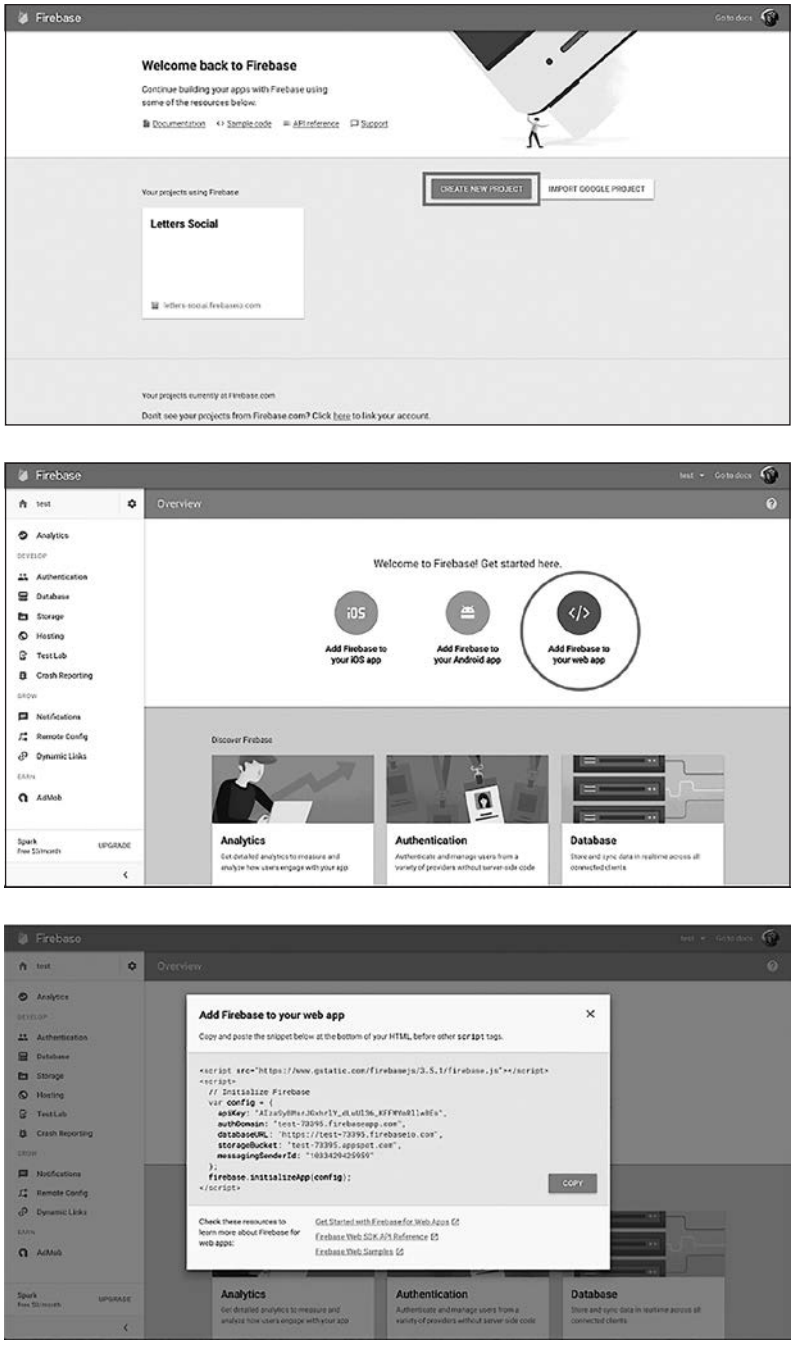


Рис. 8.2. Консоль Firebase. Создайте новый проект, который будет применяться с вашим экземпляром приложения Letters Social

Разработав проект и получив доступ к его конфигурации, можете начать работу. Firebase SDK уже установлен с демонстрационным кодом приложения, поэтому создайте новый файл с именем `core.js` в каталоге `backend`, находящемся в папке `src` (`src/backend/core.js`). В листинге 8.11 показано, как вы будете настраивать файл `core.js`, используя значения из конфигурации приложения. Я добавил в исходный код публичный ключ API Firebase, чтобы вы могли запускать приложение без регистрации, но, если хотите заменить его собственным ключом, измените соответствующие значения в каталоге конфигурации.

Листинг 8.11. Настройка серверной части Firebase (src/backend/core.js)

```
import firebase from 'firebase';

const config = {
  apiKey: process.env.GOOGLE_API_KEY,
  authDomain: process.env.FIREBASE_AUTH_DOMAIN
};

try {
  firebase.initializeApp(config);
} catch (e) {
  console.error('Error initializing firebase — check your source code');
  console.error(e);
}

export { firebase };
```

Значения вводит Webpack, если хотите задействовать собственные — измените их в каталоге конфигурации

Инициализация Firebase со своими учетными данными

Экспорт сконфигурированного экземпляра Firebase для использования в других случаях

Поскольку для аутентификации будет применяться Firebase, следует написать код, который позволит воспользоваться этой функциональностью. Чтобы начать работу, выберите платформу для аутентификации (рис. 8.3). GitHub, Facebook, Google или Twitter позволят войти пользователям, у которых уже есть одна из этих учетных записей, не требуя использовать другую комбинацию имени пользователя/логина. Предлагаю выбрать GitHub, потому что вы и большинство людей, которые увидят приложение, скорее всего, имеют учетные записи GitHub. Разумеется, можно работать также с одной или несколькими другими платформами. В рассматриваемых примерах я применяю GitHub для простоты. Выбрав платформу, следуйте инструкциям, чтобы настроить ее.

После того как вы настроили платформу по своему выбору для применения с Firebase, нужно написать еще немного кода, который позволит взаимодействовать с firebase для авторизации пользователя в системе. Firebase поставляется со встроенными инструментами, выполняющими аутентификацию с помощью различных социальных платформ. Как упоминалось, я буду использовать GitHub, вы же можете выбрать любого другого провайдера или провайдеров, которых настроили самостоятельно. Все они работают по единому шаблону (например, создают объект провайдера, настраивают область действия и т. д.). Подробнее об услугах аутентификации, предлагаемых Firebase, можно узнать на странице firebase.google.com/docs/auth/. Листинг 8.12 отражает настройку утилит аутентификации в файле `src/backend/auth.js`. Вы напишете функции для получения пользователя и токена, а также авторизации и выхода из аккаунта.

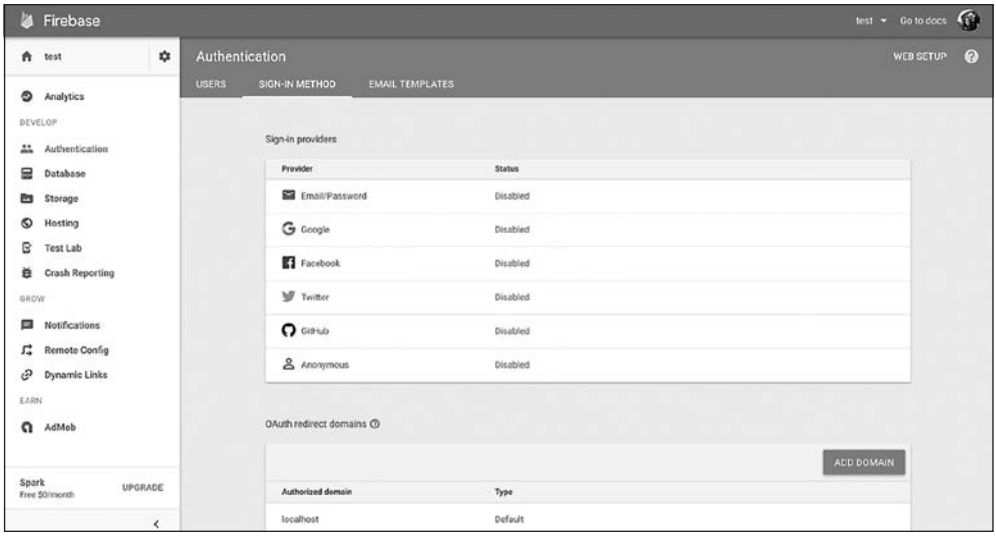


Рис. 8.3. Настройка метода аутентификации с помощью Firebase. Перейдите в раздел Authentication (Аутентификация) и выберите любого из социальных провайдеров. Затем следуйте инструкциям для этого социального аутентификатора и убедитесь, что Firebase имеет доступ к корректным учетным данным для аутентификации на указанной платформе

Листинг 8.12. Настройка инструментов аутентификации (src/backend/auth.js)

```
import { firebase } from './core';

const github = new firebase.auth.GithubAuthProvider();
github.addScope('user:email');

export function logUserOut() {
  return firebase.auth().signOut();
}

export function loginWithGithub() {
  return firebase.auth().signInWithPopup(github);
}

export function getFirebaseUser() {
  return new Promise(resolve =>
  firebase.auth().onAuthStateChanged(user =>
  resolve(user)));
}

export function getFirebaseToken() {
  const currentUser = firebase.auth().currentUser;
  if (!currentUser) {
    return Promise.resolve(null);
  }
  return currentUser.getIdToken(true);
}
```

Импортирование библиотеки Firebase, которую вы сконфигурировали

Использование Firebase для настройки поставщика аутентификации GitHub

Создание функции, которая обертывает метод выхода Firebase

Создание простой утилиты loginWithGithub, которая возвращает действие аутентификации Firebase Promise

Создание метода-обертки, чтобы получить пользователя Firebase

Позже вам понадобится токен, поэтому создается метод, который поможет получить его

Теперь, когда все готово и настроено, можно сформировать компонент, который будет обрабатывать авторизацию. Создайте файл `src/pages/Login.js`. Мы разработаем простой компонент, который сообщает пользователю, как он может авторизоваться в приложении Letter Social. В листинге 8.13 показан компонент страницы авторизации.

Листинг 8.13. Компонент Login (src/pages/Login.js)

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

import { history } from '../history';
import { loginWithGithub } from '../backend/auth';
import Welcome from '../components/welcome/Welcome';

export class Login extends Component {
  constructor(props) {
    super(props);
    this.login = this.login.bind(this);
  }
  login() {
    loginWithGithub().then(() => {
      history.push('/');
    });
  }
  render() {
    return (
      <div className="login">
        <div className="welcome-container">
          <Welcome />
        </div>
        <div className="providers">
          <button onClick={this.login}>
            <i className={'fa fa-github'} />
              log in with Github
          </button>
        </div>
      </div>
    );
  }
}

export default Login;
```

Импортирование необходимых библиотек

Создание и привязка метода авторизации

Использование метода-обертки, который вы разработали ранее, для авторизации в системе с помощью GitHub

Рендеринг компонента Welcome (включен в исходный код) или чего-то еще на свое усмотрение

Убедитесь, что метод login будет вызываться, когда пользователь нажимает кнопку авторизации

Подтверждение того, что пользователь авторизовался в системе. Ваша последняя задача — убедиться, что неаутентифицированный пользователь перенаправлен на страницу авторизации в системе. Для текущего состояния вашего приложения не будет иметь большого значения, авторизовался пользователь или вышел, потому что он может видеть только фиктивные данные, которые не связаны ни с чем в реальной жизни. Но в рабочей ситуации вполне вероятно требование, чтобы пользователь мог видеть данные, только если у него есть учетная запись и он авторизовался в системе. Это основное требование почти для всех веб-приложений, и хотя мы не будем фокусироваться на безопасности, но должны убедиться: пользователь видит социальную сеть, только если он авторизовался в системе.

Существуют различные подходы к обеспечению этой возможности. В надежных и развитых инструментах, таких как `React Router`, есть *перехватчики*, которые можно задействовать при навигации по определенному маршруту и таким образом проверить, авторизовался ли пользователь и может ли он продолжать. Это только один подход, и в вашем компоненте `Router` нет кода перехватчиков, но можно добавить некоторую логику в основной файл (`index.js`), чтобы проверить наличие пользователей и определить, по какому маршруту их следует отправить. Вы начнете работать с `React Router` и этими перехватчиками в следующих главах. А еще необходимо добавить компонент `Login` к своему роутеру.

Упражнение 8.3. Альтернативы Firebase

В этой книге мы используем Firebase в качестве «серверной части как службы». Это значительно упрощает работу в ходе обучения, но не обязательно таким же может быть подход к работе в команде. Не углубляясь: как вы думаете, что займет место Firebase в вашем приложении?

Когда пользователь авторизуется, нужно убедиться, что он также зарегистрирован в вашем API. Мы используем Firebase для аутентификации, но вы по-прежнему хотите хранить информацию о пользователях, чтобы они могли создавать сообщения и комментарии и «лайкать» понравившиеся публикации (функциональности комментариев и лайков добавим в последующих главах). Вам придется проверить, существует ли пользователь, и, если нет, создать его как пользователя в своей системе. Логика аутентификации, которую вы построите, будет учитывать все это. Мы также немного изменим код функции прослушивателя истории браузера, чтобы она могла перенаправлять людей на основании того, авторизовались они в системе или нет.

В листинге 8.14 показано, как добавить эту логику и изменить прослушиватель истории в основном индексном файле приложения (`src/index.js`).

Листинг 8.14. Добавление контейнера Login в Router (src/index.js)

```
export const renderApp = (state, callback = () => {}) => {
  render(
    <Router {...state}>
      <Route path="" component={App}>
        <Route path="/" component={Home} />
        <Route path="/posts/:postId" component={SinglePost} />
        <Route path="/login" component={Login} />
        <Route path="*" component={NotFound} />
      </Route>
    </Router>,
    document.getElementById('app'),
   callback
  );
};

let state = {
  location: window.location.pathname,
```

Добавление страницы авторизации к маршрутам

```
  user: {
    authenticated: false,
    profilePicture: null,
    id: null,
    name: null,
    token: null
  }
};

renderApp(state);

history.listen(location => {
  const user = firebase.auth().currentUser;
  state = Object.assign({}, state, {
    location: user ? location.pathname : '/login'
  });
  renderApp(state);
});

firebase.auth().onAuthStateChanged(async user => {
  if (!user) {
    state = {
      location: state.location,
      user: {
        authenticated: false
      }
    };
    return renderApp(state, () => {
      history.push('/login');
    });
  }
  const token = await getFirebaseToken();

  const res = await API.loadUser(user.uid);

  let renderUser;

  if (res.status === 404) {
    const userPayload = {
      name: user.displayName,
      profilePicture: user.photoURL,
      id: user.uid
    };
    renderUser =
      await API.createUser(userPayload).then(res => res.json());
  } else {
    renderUser = await res.json();
  }
  history.push('/');
  state = Object.assign({}, state, {
    user: {
      name: renderUser.name,
      id: renderUser.id,
```

Слежение за пользователем и соответственно обновление созданного объекта состояния

В слушателе истории сначала проверяется, существует ли такой пользователь Firebase

Использование функции async для реагирования на изменение состояния пользователя Firebase

Если пользователя нет, обновляется состояние и рендерится приложение

Если пользователь есть, его токен извлекается с помощью команды await и утилиты Firebase, которую вы создали

Попытка загрузить пользователя из API

Объявление пользовательской переменной для присвоения

Если пользователя нет, нужно его зарегистрировать

Создание полезной нагрузки пользователя, которую поймет ваш API

Отправьте запрос API и используйте ответ

Если пользователь уже существует, он применяется для рендеринга приложения

Перемещение пользователя на главную страницу

Обновление состояния приложения

```
      profilePicture: renderUser.profilePicture,
      authenticated: true
    },
    token
  });
  renderApp(state);
});
```

Рендеринг приложения с новым состоянием

Теперь пользователи могут авторизоваться и получить учетную запись, созданную для них на лету. Вы должны обновить навигационную панель, чтобы они знали, как это сделать, а также могли видеть команду выхода из системы. Возможно, вы помните, что ранее в данной главе передавали свойство `user` компоненту `Navbar`, хотя его еще не существовало. Теперь, когда этот компонент существует, он может отображать разные представления в зависимости от состояния аутентификации. В листинге 8.15 показано, как внести эти изменения в компонент `Navbar`.

Листинг 8.15. Обновление компонента Navbar (src/components/nav/navbar.js)

```
import React from 'react';
import PropTypes from 'prop-types';

import Link from '../router/Link';
import Logo from './logo';
import { logUserOut } from '../../backend/auth';

  export const Navigation = ({ user }) => (
  <nav className="navbar">
    <Logo />
    {user.authenticated ? (
      <span className="user-nav-widget">
        <span>{user.name}</span>
        <img width={40} className="img-circle"
        src={user.profilePicture} alt={user.name} />
        <span onClick={() => logUserOut()}>
          <i className="fa fa-sign-out" />
        </span>
      </span>
    ) : (
      <Link to="/login">
        <button type="button">Log in or sign up</button>
      </Link>
    )}
  </nav>
);

Navigation.propTypes = {
  user: PropTypes.shape({
    name: PropTypes.string,
    authenticated: PropTypes.bool,
    profilePicture: PropTypes.string
  }).isRequired
};

export default Navigation;
```

Если пользователь аутентифицирован, отображается информация о его профиле (имя, изображение профиля)

Допуск пользователя к опции выхода из системы (с использованием ранее созданной утилиты Firebase)

Если пользователь не авторизован, выводится ссылка для авторизации

Объявление proptypes

Экспортирование компонента для использования

8.3. Резюме

В этой главе вы начали работать с собственноручно созданным компонентом `Router`, добавили в приложение еще несколько компонентов, связанных с маршрутизацией, выполнили определенную реорганизацию и добавили аутентификацию пользователей с Firebase. Вот некоторые моменты, которые стоит отметить.

- Firebase — это инструмент «серверная часть как служба», который позволяет аутентифицировать пользователей, хранить данные и многое другое. Он способен выполнить многое, не требуя вести какие-либо разработки на сервере, и это отличное место для многих пользовательских проектов.
- Вы можете интегрировать в свой роутер API истории браузера. А также создавать компоненты `Link`, которые не требуют полной перезагрузки страницы, вместо обычных элементов привязки.
- Firebase способен обрабатывать аутентификацию и данные сеанса пользователя автоматически. Мы изучим более сложные методы обработки изменяющегося состояния в последующих главах, когда будем рассматривать Flux, Redux и даже применение Firebase на сервере для рендеринга на стороне сервера.

Тестирование — очень важный этап разработки хорошего программного обеспечения. В следующей главе рассмотрим тестирование компонентов React с помощью Jest и Enzyme.

9 Тестирование компонентов React

- Тестирование клиентских приложений.
- Настройка тестирования для React.
- Тестирование компонентов React.
- Настройка полноты тестирования.

В предыдущей главе вы добавили в приложение несколько важных функций. Теперь в нем есть маршрутизация и пользовательское состояние, и вы разбили его на более мелкие части. Вы даже добавили базовую аутентификацию, чтобы пользователи могли войти в систему, используя свой профиль GitHub. Приложение уже выглядит более законченным, даже если это не волнует никого в Facebook или Twitter. Сейчас можно добиться намного большего с помощью React, чем было в начале. Но поскольку мы сосредоточились на изучении основ, то опустили важную часть процесса разработки — *тестирование*.

Я не рассматривал тестирование в начале книги, чтобы избавить вас от перегрузки при изучении React и основ тестирования одновременно. Но это не означает, что эта часть веб-разработки не важна. В данной главе сосредоточимся на тестировании — фундаментальном участке разработки высококачественных программных решений. Вместо того чтобы демонстрировать тесты для каждого из компонентов, рассмотрим демонстрационный пример, чтобы вы разобрались в важных принципах работы и смогли писать тесты сами.

К концу этой главы вы поймете некоторые основные принципы тестирования веб-приложений. А также настроите тесты и исполнитель тестов, поработаете с Jest, Enzyme и тестовым рендером React и научитесь использовать общие инструменты тестирования. Научившись тестировать свои приложения, вы добавите еще один уровень уверенности к навыкам разработки React-проектов.

Получение исходного кода

Как и прежде, вы можете получить исходный код примеров из этой главы, перейдя в репозиторий GitHub по адресу github.com/react-in-action/letters-social. Если планируете начать работу здесь самостоятельно с нуля, возьмите исходный код примеров

из глав 5 и 6 (если изучили их и сами выполнили примеры) или обратитесь к ветви, относящейся к данной главе (chapter-9).

Помните, что каждая ветвь содержит итоговый код главы (например, chapter-7-8 содержит код, получаемый в конце глав 7 и 8). Вы можете выполнить в оболочке командной строки одну из следующих команд по своему выбору, чтобы получить код примеров из текущей главы. Если репозитория вообще нет, выполните такую команду:

```
git clone git@github.com:react-in-action/letters-social.git
```

Если у вас уже клонирован репозиторий, то следующую:

```
git checkout chapter-7-8
```

Возможно, вы перешли сюда из другой главы, поэтому стоит проверить, установлены ли у вас все нужные зависимости, с помощью следующей команды:

```
npm install
```

Тестирование при разработке программного обеспечения — это процесс проверки предположений. Например, вы создаете приложение (такое как Medium, Ghost или WordPress), которое позволяет пользователям писать и размещать записи в блогах. Пользователи вносят ежемесячную плату и получают хостинг и инструменты для запуска собственного блога. Разрабатывая пользовательский интерфейс приложения, нужно обеспечить несколько *обязательных* для него ключевых моментов (среди прочего), включая правильное отображение сообщений и разрешение пользователям редактировать их.

Как удостовериться, что приложение делает то, что нужно? Можете зайти туда и посмотреть, работает ли оно. Щелкайте кнопкой мыши, вводите данные и используйте приложение как можно большим количеством способов. Этот ручной процесс довольно хорош и представляет собой первую линию защиты от ошибок и регрессии. Вы всегда должны следить за тем, над чем работаете, однако проверить интерфейс быстро и качественно непросто.

По мере развития приложения количество ситуаций и особенностей, которые требуется проверять вручную, увеличивается с невероятной скоростью. Я работал над приложениями с тысячами тестов, но есть много приложений, где их намного больше. На момент написания книги библиотека React содержала 4855 тестов. Нет никаких шансов, что кто-то, решив протестировать React, сможет проверить вручную предположения, связанные со всеми этими тестами.

К счастью, вместо этого вы можете применить программное обеспечение для тестирования программного обеспечения. Компьютеры превосходят нас по крайней мере в двух важных областях: скорости и системности. Мы можем использовать программное обеспечение для тестирования кода такими способами, которые никогда не смогли бы выполнить вручную, даже если бы имели целую армию помощников. Возможно, вы думаете: «Мой проект маленький и очень простой — мало что может пойти не так». Но даже если ваши навыки кодирования превосходны, ошибки неизбежны. Приложения будут сбоить и работать непредсказуемым образом, когда вы меняете что-то (а иногда — даже если вы этого не делаете).

Но вместо того, чтобы отчаиваться из-за неизбежности ошибок, мы можем смириться с тем, что они возникнут, и предпринять шаги, чтобы свести к минимуму их влияние и частоту. Вот где начинается тестирование. Возможно, вы имеете общее представление о том, что такое тестирование, но для начала следует изучить различные его типы. Имейте в виду, что мир тестирования огромен и я не могу охватить даже его часть. Я не буду глубоко освещать тестирование как область программирования. И не стану углубляться в некоторые типы тестирования, включая интеграционное тестирование, регрессионное тестирование, автоматизацию тестирования и др. Но к концу главы вы должны знать достаточно, чтобы начать тестирование компонентов React несколькими способами.

9.1. Типы тестирования

Как уже говорилось, тестирование программного обеспечения — это процесс применения программного обеспечения для проверки ваших предположений. Поскольку вы применяете программное обеспечение для тестирования программного обеспечения, в итоге будете использовать те же примитивы, что и при создании программного обеспечения: булевы значения, числа, строки, функции, объекты и т. п. Важно помнить, что здесь нет никакой магии — просто больше кода.

Существуют различные типы тестирования, и вы будете задействовать некоторые из них, чтобы протестировать React-приложение. Они охватывают различные функции приложения, а при совместном использовании и в правильных пропорциях должны обеспечить значительную степень уверенности в вашем приложении. Различные типы тестов касаются разных частей и областей применения. Хорошо протестированное приложение проверит отдельные единицы функциональности, составляющие основные части приложения. Оно проверит коллекции единиц функциональности, а также — на самом высоком уровне — точки, в которых все объединяется (например, пользовательский интерфейс).

Вот несколько типов тестирования.

- *Блочное тестирование* — тесты блоков сосредоточены на отдельных единицах функциональности. Например, у вас есть метод утилиты для получения новых сообщений с сервера. Блочный тест будет сосредоточен лишь на одной функции — ничто другое его не волнует. Подобно компонентам, такие тесты позволяют проводить реорганизацию и повышать модульность.
- *Тестирование сервисов* — данное тестирование сосредоточено на наборах функциональности. Эта часть спектра тестов может включать в себя множество дробных частей и фокусов. Дело в том, что вы тестируете код, который находится не на самом высоком (см. далее интеграционные тесты) или самом низком уровне функциональности. Пример сервисного теста может быть чем-то вроде инструмента, который использует несколько единиц функциональности, но сам находится не на уровне теста интеграции.
- *Тестирование интеграции* — сосредоточено на еще более высоком уровне тестирования — интеграции различных частей приложения. Такие тесты проверяют,

как работают совместно сервисы и функции более низкого уровня. Как правило, эти тесты работают с приложением через его пользовательский интерфейс, а не через отдельный код, стоящий за ним. Они могут имитировать щелчки кнопкой мыши, пользовательский ввод и другие взаимодействия, которые управляют приложением.

Возможно, вам интересно, на что похожи эти тесты в коде. Мы вскоре займемся этим, но сначала нужно поговорить о том, как они работают вместе при общем подходе к тестированию. Если раньше вы уже проводили тестирование, возможно, слышали о *пирамиде тестирования*. Эта пирамида (рис. 9.1) обычно показывает пропорции различных типов тестов, которые требуется написать. В этой главе вы будете создавать только единичные тесты для своих компонентов.

Рис. 9.1. Пирамида тестирования — это способ определения количества и типов тестов, которые вы пишете при тестировании своих приложений. Обратите внимание на то, что некоторые типы тестов занимают больше времени, а следовательно, более затратны с точки зрения времени (и финансов)

Зачем тестировать? Существуют парадигмы разработки программного обеспечения, где тестирование является первоочередной задачей всего процесса разработки. Это означает, что тестирование важно, рассматривается в начале и на протяжении всего процесса разработки и обычно играет роль в определении того, когда что-то считается завершенным. Разумеется, консенсус в том, что тестирование — отличный процесс для разработки программного обеспечения, но есть определенные парадигмы, где оно играет центральную роль. Возможно, вы слышали о *разработке под управлением тестирования* (TDD). Когда практикуется TDD, то, как следует из названия, сам процесс написания программного обеспечения управляется тестированием. В ходе его разработчик обычно создает провальный тест (он вводит утверждения, которые еще не были выполнены), пишет достаточно кода, чтобы его пройти, реорганизует какое-либо дублирование, а затем переходит к следующей функции и повторяет процесс.

Вам не нужно практиковать исключительно TDD для написания отличного программного обеспечения, но прежде, чем читать дальше, подумайте о некоторых его преимуществах. Если вы хорошо разбираетесь в результатах тестирования, не стесняйтесь перейти к следующему разделу, где мы начнем с тестирования в React. Но я хочу задать важный вопрос: почему мы вообще тестируем?

Во-первых, мы хотим написать работающее программное обеспечение. В современном программном обеспечении так много взаимосвязанных частей, что было бы глупо ожидать, что все они постоянно будут надежно работать. Неизбежно что-то выйдет из строя, и лучше предположить, что некоторые функции окажутся неудачными, чем что они все время будут работать. Мы можем свести к минимуму то, что способно повредить наше программное обеспечение, проверяя свои предположения. Тестирование заставляет вас сформулировать ожидания от собственного программного обеспечения (или пересмотреть их). Вы проведете его для различных ситуаций, с которыми оно способно справиться, и убедитесь, что оно правильно их обрабатывает.

Во-вторых, тестирование своего программного обеспечения помогает вам лучше писать код. Написание тестов побуждает вас задуматься над тем, какие функции выполняет код, особенно если выполняет их заранее (как в TDD). Гораздо менее предпочтительный вариант — написать тесты после создания кода, но все же это лучше, чем полное отсутствие тестов. Проведение тестирования поможет лучше понять свой код и подтвердить предположения, которые вы и другие делаете о том, как все работает.

В-третьих, интеграция тестирования в рабочий процесс разработки программного обеспечения означает, что вы способны выдавать код чаще. Возможно, прежде вы слышали, как люди, работающие в технологической сфере, упоминают «отгрузку чаще». Обычно это означает выпуск программного обеспечения с нарастающей частотой. В прошлом компании зачастую выпускали программное обеспечение только после длительной работы над ним и лишь несколько раз в год (или по крайней мере довольно редко).

Сейчас мышление изменилось и люди поняли, что поэтапная итерация полезна для программного обеспечения: вы можете быстрее получить отзывы от пользователей и других заинтересованных лиц, проще становится экспериментировать и др. Уверенность в надежности хорошо протестированного приложения является ключевой частью этого процесса. Используя *непрерывную интеграцию* (CI) или инструменты *непрерывного развертывания*, такие как Circle CI (circleci.com), Travis CI (travis-ci.org) или другие, вы можете сделать тестирование частью процесса развертывания программного обеспечения. Смысл таков: *если тесты проходят, ПО развертывается*. Эти инструменты обычно запускают тесты в чистой среде и, если они проходят удачно, отправляют код в любую систему, запускающую приложение. На рис. 9.2 показан процесс, который приложение Letters Social задействует для тестирования и развертывания.

Наконец, тесты помогают: вы возвращаетесь и переделываете код или перемещаете его. Скажем, ваши требования меняются и требуется переместить некоторые компоненты. Если вы сохранили компоненты модульными и они удачно прошли

тестирование, их перемещение должно оказаться легким. Конечно, можно перемещать нетестированный код, но так вы будете гораздо менее уверены в том, что он не повредит другие части системы, чем когда код протестирован.

Рис. 9.2. Последовательность развертывания Letters Social. Процесс сборки CI запускается, когда я (или кто-то, кто вносит вклад в репозиторий) вставляю код. Провайдер CI (в данном случае Circle) использует контейнеры Docker для быстрого и надежного запуска тестов. Если тесты будут пройдены, код будет развернут в любой службе, которую вы применяете для запуска кода. В нашем случае это Now

О преимуществах и теории тестирования в программном обеспечении можно рассказывать долго, но это выходит за рамки книги. Если хотите узнать больше, посмотрите книги *The Art of Unit Testing, Second Edition* (Manning Publications, 2013) Роя Ошерове (Roy Osherove) и *Growing Object-Oriented Software: Guided by Tests* Ната Приса (Nat Pryce) и Стива Фримена (Steve Freeman) (Addison-Wesley, 2009).

9.2. Тестирование компонентов React с помощью Jest, Enzyme и React-test-renderer

Программное обеспечение для тестирования — это просто дополнительный софт, выполненный из тех же примитивов и базовых элементов, что и обычные программы, хотя люди разработали специальные инструменты, помогающие в процессе тестирования. Вы можете попытаться написать необходимые средства для запуска своих тестов, но сообщество специалистов, создающих программы с открытым исходным кодом, уже выполнило огромный объем работы и разработало огромное количество мощных инструментов, поэтому проще будет использовать их.

Для тестирования React-приложений вам понадобятся несколько типов библиотек.

- *Исполнитель тестов* — вам нужно что-то, чтобы выполнить тесты. Большинство тестов могут быть созданы как обычные файлы JavaScript, но вы захотите воспользоваться некоторыми добавленными функциями исполнителей тестов, например запускать более одного теста за раз и сообщать об ошибках или успешном завершении.

В этой книге для большинства видов тестирования вы будете использовать Jest. Это тестовая библиотека, разработанная программистами из компании Facebook. Можете рассмотреть и популярные альтернативы с меньшим количеством встроенных функций — Mocha (mochajs.org) и Jasmine (jasmine.github.io). Jest часто применяется для тестирования React-приложений, но для других фреймворков также создаются адаптеры. Исходный код включает установочный файл (`test/setup.js`), который вызывает адаптер для React.

- *Дублирование тестов*. При написании тестов вы стараетесь по максимуму избегать их связывания с другими уязвимыми или непредсказуемыми частями инфраструктуры; другие инструменты, на которые вы полагаетесь, должны быть смоделированы — заменены «поддельной» функцией, которая ведет себя так, как от нее ожидают. Это помогает сосредоточиться на тестируемом коде и модульности, потому что тесты не привязаны к точной структуре кода в данный момент времени. Для моделирования и двойного тестирования используйте Jest, есть и другие библиотеки, которые делают то же самое, например Sinon (sinonjs.org).
- *Библиотеки суждений*. Можно задействовать JavaScript, чтобы добавить утверждения о своем коде (например, X равно Y?). Но есть много крайних случаев, которые следует учитывать. Разработчики создали решения, облегчающие написание утверждения о вашем коде. Jest поставляется со встроенными методами утверждения, поэтому вы будете полагаться на них.
- *Помощники по окружению*. Тестирование кода, который должен запускаться в среде браузера, предъявляет несколько иные требования. Среда браузера уникальна и включает DOM, пользовательские события и прочие обычные части веб-приложений. Рассматриваемые инструменты тестирования помогут успешно эмулировать среду браузера. Вы будете использовать Enzyme и React test renderer при тестировании компонентов React. Enzyme облегчает тестирование. Он обеспечивает надежный API, который позволяет запрашивать различные типы компонентов и элементов HTML, устанавливать и получать свойства компонентов, проверять и определять состояние компонентов и многое другое. React test renderer выполняет аналогичные действия, а также может создавать моментальные снимки компонентов. Мы не будем углубляться во все особенности API Enzyme или React test renderer, но не стесняйтесь узнать больше по адресу www.airbnb.io/enzyme и www.npmjs.com/package/reacttest-renderer.
- *Библиотеки, специфичные для конкретного фреймворка*. Существуют библиотеки, созданные специально для React (или других фреймворков), которые упрощают написание тестов для определенного фреймворка. Обычно эти абстракции разрабатывают для того, чтобы помочь в тестировании библиотеки или фреймворка и справиться с настройкой всего, что необходимо для последнего. В React почти все — «просто JavaScript», поэтому даже в этих инструментах есть немного «магии».
- *Средства покрытия*. Благодаря детерминированному характеру кода люди нашли способы определения того, какие его части «покрываются» тестами. Это здорово, потому что вы получите показатель, который служит ориентиром

при определении того, насколько хорошо протестирован код. Это не подменяет логику и базовый анализ (100%-ный охват кода не означает, что не возникнет ошибок), но поможет проверить код. Вы примените встроенный инструмент охвата Jest, в котором используется популярный инструмент под названием Istanbul (github.com/gotwarlost/istanbul).

Начните с установки инструментов, которые станете использовать для тестов. Если вы клонировали репозиторий книг из GitHub, эти инструменты уже должны быть установлены. При смене глав обязательно выполните команду `npm install`, чтобы убедиться: у вас есть все библиотеки для данной главы.

9.3. Написание первых тестов

После того как установлены нужные инструменты, можно приступать к работе. В этом разделе настроим команды для запуска тестов и начнем тестирование некоторых базовых компонентов React. Вы будете делать предположения по поводу определенных компонентов и смотреть, каковы результаты их тестирования.

Но прежде я должен обратить внимание на несколько моментов относительно Jest и на то, где будет работать код ваших тестов. Jest может быть настроен для работы в разных средах в зависимости от типа тестов, которые вы пишете. Если вы создаете тесты для React-приложений, запускаемых в браузере, то должны сказать Jest, чтобы он обеспечил виртуальную среду браузера, необходимую для правильной эмуляции реального браузера. Jest использует для этого библиотеку `jsdom`. Если вы пишете тесты для приложений node.js, вам не нужны дополнительная память и нагрузка среды `jsdom` — вы просто хотите протестировать серверный код. Jest настроен на запуск по умолчанию тестов, ориентированных на браузер, поэтому переопределять что-либо не требуется.

Упражнение 9.1. Обзор типов тестирования

Существует несколько типов тестирования. Попробуйте сопоставить название типа с его описанием.

- Блок.
- Служба.
- Интеграция.

❑ Сложные, часто нестабильные тесты требуют много времени для написания и работы. Они проверяют, как разные системы работают вместе на высоком уровне. Зачастую таких тестов бывает меньше, чем других.

❑ Более простые тесты, которые проверяют работу конкретной системы, не взаимодействующей с другими.

❑ Низкоуровневые узконаправленные тесты, которые фокусируются на исследовании небольших фрагментов функциональности. В наборе их должно быть больше всего.

9.3.1. Начало работы с Jest

Для запуска тестов, как уже сказано, используйте Jest. Вы можете запустить Jest из командной строки, и он выполнит ваши тесты, так что добавьте сценарий в файл `package.json`, чтобы его можно было запускать. В листинге 9.1 показано, как добавить пользовательский сценарий в файл `package.json`. Если вы клонировали репозиторий из GitHub, этот сценарий должен быть доступен.

Листинг 9.1. Настройка пользовательского сценария npm (package.json)

```
{
  //...
  "scripts": {
    //...
    "test": "jest --coverage",
    "test:w": "jest –watch --coverage",
    "jest": {
      "testEnvironment": "jsdom",
      "setupFiles": ["raf/polyfill", "./test/setup.js"]
    },
    "repository": {
      "type": "git",
      "url": "git+ssh://git@github.com/react-in-action/letters-social.git"
    },
    "author": "Mark Thomas <hello@ifelse.io>",
// ...
```

Запуск тестов и их покрытие

Запуск тестов в режиме просмотра

Настройка Jest. В пример кода включены некоторые тестовые помощники и заглушки

Теперь, когда у вас есть команда для запуска тестов (`npm test`), приступайте. Вы еще не должны были получать какую-то полезную информацию, потому что нет тестов для запуска (Jest должен предупредить вас об этом). Вы также можете запустить `npm run test: w`, чтобы Jest работал в режиме просмотра. Это полезно, если вы не хотите каждый раз запускать тесты вручную. Режим многонаправленного просмотра Jest делает его особенно полезным для работы — он обеспечит запуск только тестов, связанных с измененными файлами. Это полезно, если набор тестов невелик и вы не хотите каждый раз запускать их один за одним. А еще можете задать шаблоны регулярных выражений или поиск по текстовой строке для запуска определенных тестов.

Отладка имеет значение

Когда дело доходит до оценки библиотек, их тестированию (и даже тестированию в целом) иногда уделяют внимание в последнюю очередь. Это неправильно по крайней мере по двум причинам. Из-за непригодных для тестирования библиотек продажи кода могут упасть и даже полностью прекратиться, что крайне невыгодно для команды-разработчика. Это приводит к созданию менее стабильного кода, который сложнее поддерживать и с которым в целом трудно работать.

Еще одним недостатком является то, что, если вы или ваша команда тратите много времени на написание тестов, ваши усилия могут существенно отразиться на вашей занятости. Это может быстро привести к потере денег бизнесом, потому что его разработчики тратят больше времени для выполнения работы, которую им нужно сделать. Мне приходилось наблюдать и то и другое. Если тестирование с самого начала не считалось основным приоритетом, с течением времени оно становилось все более трудным и рассматривалось как работа на один день. Результатом становился код, который с трудом можно было изменить, потому что ожидаемая функциональность больше не подтверждалась тестами.

Еще одна причина относиться к инструментам тестирования как к важной части процесса заключается в том, что, если вы действительно протестируете свой код, это поможет сэкономить время. Если у вас плохие или долго работающие тесты, на их выполнение ежедневно может затрачиваться много времени. Магического решения этой проблемы нет, но отладка и настройка инструментов тестирования в качестве первоочередных мер могут помочь вам в долгосрочной перспективе.

9.3.2. Тестирование функционального компонента без состояния

Пора начинать писать тесты. Вначале сосредоточимся на относительно простом примере тестирования компонента. Вы собираетесь протестировать компонент `Content`. Его функция заключается в простой поддержке рендеринга абзаца и его содержимого. В листинге 9.2 показана структура компонента.

Листинг 9.2. Компонент Content (src/components/post/Content.test.js)

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

const Content = (props) => {
  const { post } = props;
  return (
    <p className="content">
      {post.content}
    </p>
  );
};

Content.propTypes = {
  post: PropTypes.object,
};
export default Content;
```

Компонент принимает объект свойств post и использует свойство content post для рендеринга элемента абзаца

Присвоение абзацу класса content

Внутреннее содержимое элемента абзаца является содержимым post

Компонент экспортируется — это важно, потому что вам нужно импортировать компонент в свои тесты

Первое, что вы можете сделать, когда начинаете писать тесты, — это подумать о том, какие предположения следует проверить. То есть, завершившись, тесты должны подтвердить для вас определенные моменты и стать своего рода гарантией.

На самом деле я полагаюсь на тесты, чтобы они не прошли, когда я вношу изменения в определенную функцию или часть системы. Они подтверждают предположение о том, что внесенные изменения представляют собой изменение приложения или системы. Это делает написание кода намного более комфортным, потому что, с одной стороны, есть заранее сделанная запись о том, как должны были работать функции, а с другой — можно понять, как сделанные изменения влияют на приложение в целом.

Взглянем на ваш компонент и подумаем о том, как можно его протестировать. Есть несколько предположений относительно этого компонента, которые вы хотите проверить. Во-первых, он должен отображать какое-то содержимое, переданное в качестве свойства. Во-вторых, ему необходимо назначить имя класса элементу абзаца. Помимо этого, моментов, на которых нужно сосредоточиться, немного. Этого должно быть достаточно, чтобы начать писать тест.

Вы заметите, что «React работает правильно» — это не то, что вы пытаетесь протестировать. Мы также исключили такие моменты, как «функция может быть выполнена», «компилятор JSX будет работать» и некоторые другие фундаментальные предположения об используемых технологиях. Данные моменты важны для тестирования, но тесты, которые вы пишете, никогда не смогли бы адекватно или точно подтвердить их. Эти и другие проекты отвечают за составление собственных тестов и обеспечение их работы. Это подчеркивает важность выбора надежного, проверенного и постоянно обновляемого программного обеспечения. Если у вас есть серьезные сомнения в надежности React, они необоснованны. Хотя React и неидеальна, она используется в самых популярных в мире веб-приложениях, включая Facebook.com и Netflix.com. Конечно, ошибки случаются, но маловероятно, что вы столкнетесь с ними в рассматриваемой простой ситуации.

Вы знаете кое-что о компоненте, который хотите проверить, но могли бы пойти другим путем, если бы начали с нуля и сначала написали тест. Возможно, вы бы подумали: «Нам нужен компонент, который отображает содержимое, имеет определенный тип и определенное имя класса, чтобы CSS работал». Возможно, затем приступили бы к написанию теста, который подтвердил бы эти условия. Вы идете другим путем из-за того, что узнали о React, но можете заметить, что старт с теста упрощает задачу: вначале вы продумываете и планируете компонент. Как уже говорилось, разработка, основанная на тестах (TDD), учит вас размышлять, что делает предварительное написание тестов центральной частью разработки программного обеспечения.

Посмотрим, как протестировать компонент. Для этого нужно написать *набор* (группу) тестов. Индивидуальные тесты делают *утверждения* (утверждения о коде, которые могут быть истинными или ложными) для проверки допущений. Например, тест для вашего компонента будет *утверждать*, что настроено правильное имя класса. Если какое-либо из утверждений не подтверждается, тест потерпит неудачу. Так вы узнаете, что в вашем приложении что-то неожиданно изменилось или больше не работает. В листинге 9.3 показано, как настроить каркас теста.

Обратите внимание на то, что имя файла для компонента заканчивается на `.test.js`. Это соглашение, которого вы можете придерживаться, если хотите. Jest будет искать файлы, которые заканчиваются на `.spec.js` или `.test.js`, и запускать

эти тесты по умолчанию. Если вы решите следовать другому соглашению, нужно явно указать Jest, какие файлы вы хотите запустить, добавив их в вызов командной строки (`jest --watch ./my.cool.test.file.js`, например). Вы будете выполнять соглашение `.test.js` для всех своих тестов.

Также хорошо бы заметить, где находятся тестовые файлы. Некоторые предпочитают размещать все свои тесты в зеркальном каталоге `test`, обычно находящемся в корневой директории проекта. Для каждого тестируемого файла они создают соответствующий файл в тестовом каталоге. Это прекрасный способ структурировать информацию, но вы можете обнаружить тестовые файлы и рядом с исходными файлами. Применяйте этот метод (хотя хорош любой подход).

Листинг 9.3. Каркас теста для компонента Content (src/components/post/Content.test.js)

```
import React from 'react';
import { shallow } from 'enzyme';
import renderer from 'react-test-renderer';

import { Content } from './Content';

describe('<Content/>', () => {
  test('should render correctly', () => {

  });
});
```

Импортирование React

Импортирование связанных вспомогательных методов

Импортирование компонентов для тестирования

Jest использует методы в стиле Jasmine (jasmine.github.io), подобные описанию групповых тестов

Фактический тест — функция it — также предоставляется Jest глобально

Возможно, вы заметили, что пока ничего особенного в функциях `describe` нет. Они предназначены в первую очередь для организации и обеспечения возможности разделить тесты на соответствующие звенья, чтобы протестировать разные части кода. Может показаться, что в таком маленьком файле в этом нет большой надобности, но я работал с тестовыми файлами длиной 2000–3000 строк (и даже больше) и могу утверждать: удобочитаемые тесты помогают хорошо выполнять тестирование.

Пишите ясные тесты!

Вы когда-нибудь читали тестовый код, который не получил такого же описания, что и код, который он должен тестировать? Со мной это случалось не раз. Чтение неясного тестового кода может сбить с толку и даже разочаровать. Тесты — это всего лишь код, поэтому они все равно должны быть ясными и читабельными, верно? Я уже упоминал, что иногда тестированию уделяют значительно меньше внимания, чем написанию кода приложения. Тестовый код можно рассматривать как задачу, которая должна быть выполнена, или даже барьер между вами и кодом приложения, поэтому требования к нему снижаются. К этой тенденции легко сползти, но реальность такова, что слабые тесты могут быть такими же плохими, как и некачественно написанный код приложения. Тесты должны служить еще одной формой документации для кода, и такой, которую могут прочитать разработчики. Помните, что тестовый код должен быть ясным.

Jest будет искать файлы для тестирования, а затем выполнять функции `describe` и `it`, вызывая функции обратного вызова, которые вы им предоставили. Но что вам необходимо вложить в них? Следует настроить *утверждения*. Для этого нужно что-то утверждать. Именно здесь на сцену выходит Enzyme — он позволяет разработать пригодную для тестирования версию компонента, которую вы можете проверить, и сделать утверждения. Задействуйте метод *мелкого рендеринга* Enzyme, создающий легкую версию вашего компонента, которая не выполняет полную установку или вставку в DOM. Вам также необходимо задать некоторые *макетные* (имитирующие) данные для использования компонентом. В листинге 9.4 показано, как добавить тестовую версию компонента в свой тестовый набор. Прежде чем приступать к написанию тестов, убедитесь, что выполнили команду `npm run test:w` в оболочке командной строки, чтобы запустить исполнитель тестов.

Листинг 9.4. Мелкий рендеринг (src/components/post/Content.test.js)

```
import React from 'react';
import { shallow } from 'enzyme';
import renderer from 'react-test-renderer';

import { Content } from './Content';

describe('<Content/>', () => {
  describe('render methods', () => {
    it('should render correctly', () => {
      const mockPost = {
        content: 'I am learning to test React components',
      };
      const wrapper = shallow(<Content post={mockPost} />);
    });
  });
});
```

Создание фиктивного объекта сообщения, который компонент может использовать

Выполнение мелкого рендеринга компонента и сохранение возвращенной обертки

Теперь у вас есть установленный тестовый компонент, о котором вы можете делать утверждения. Для этого задействуйте встроенную функцию `expect()` из Jest. Если вы применяли другую библиотеку утверждений, можете использовать что-нибудь еще. Помните, что библиотеки утверждений существуют для упрощения утверждений. Например, проверка того, является ли объект *действительно равным* (то есть равно каждое из его свойств), может оказаться сложной задачей. При создании тестов вы не должны сосредотачиваться на внедрении множества новых функциональных возможностей только для того, чтобы их написать, — вы должны сфокусироваться на тестируемом коде.

Чтобы проверить рассматриваемый компонент, сделайте несколько утверждений, о которых мы говорили ранее: имя класса, внутреннее содержание и тип элемента. Также создайте тест-снимок с помощью React test renderer. *Тестирова-*

ние моментальных снимков — это функция Jest, которая позволяет протестировать вывод компонентов уникальным образом. Тестирование моментальных снимков тесно связано с визуальным регрессионным тестированием — процессом, в котором визуальный вывод приложения можно сравнить с выводом другого приложения.

Если обнаружено различие в изображениях, вы знаете, что тест не удался и нуждается в настройке или по крайней мере следует обновить выходной снимок. Вместо изображений Jest будет создавать выводы JSON для тестов и хранить их в особым образом названных каталогах. Они должны быть добавлены в систему контроля версий вместе со всем остальным кодом. В листинге 9.5 показано, как использовать Jest, Enzyme и React test renderer, чтобы сделать эти утверждения.

Листинг 9.5. Выполнение утверждений (src/components/post/Content.test.js)

```
import React from 'react';
import { shallow } from 'enzyme';
import renderer from 'react-test-renderer';

import Content from '../../../src/components/post/Content';

describe('<Content/>', () => {
  test('should render correctly', () => {
    const mockPost = {
      content: 'I am learning to test React components'
    };
    const wrapper = shallow(<Content post={mockPost} />);
    expect(wrapper.find('p').length).toBe(1);
    expect(wrapper.find('p.content').length).toBe(1);
    expect(wrapper.find('.content').text()).toBe(mockPost.content);
    expect(wrapper.find('p').text()).toBe(mockPost.content);
  });
  test('snapshot', () => {
    const mockPost = {
      content: 'I am learning to test React components'
    };
    const component = renderer.create(
      <Content post={mockPost} />);
    const tree = component.toJSON();
    expect(tree).toMatchSnapshot();
  });
});
```

Импортирование компонента, который нужно протестировать

Импортирование enzyme и react-test-renderer

Использование функции describe в стиле Jasmine, чтобы сгруппировать тесты

Создание mock-сообщения

Использование метода shallow из Enzyme для рендеринга компонента

Создание моментального снимка с применением Jest и response-test-renderer

Если исполнитель тестов запущен, вы должны увидеть результат из Jest. Инструменты командной строки Jest значительно улучшились с тех пор, как выпущен исполнитель тестов, и вы можете увидеть важную информацию о своих тестах на терминале.

9.3.3. Тестирование компонента CreatePost без Enzyme

Теперь, когда у вас есть первый работающий тест, переходите к тестированию более сложных компонентов. По большей части тестирование компонентов React должно быть простым. Если вы обнаружите, что создаете компонент, у которого тонны встроенной функциональности и, соответственно, огромные связанные тесты, можете разбить его на несколько компонентов (хотя это не всегда возможно).

Компонент `CreatePost`, который вы хотите протестировать, обладает большей функциональностью, чем компонент `Content`, и ваши тесты должны будут учесть это. В листинге 9.6 показан компонент `CreatePost`, чтобы вы могли просмотреть его, прежде чем писать для него тесты. Компонент `CreatePost` используется компонентом `Home` для запуска отправки новых сообщений. Он отображает элемент `textarea`, который обновляется, когда пользователь в нем печатает, и кнопку, отправляющую форму с данными, когда пользователь нажимает ее. При нажатии вызывается функция обратного вызова, переданная родительским компонентом. Можете проверить эти предположения и убедиться, что все работает так, как вы ожидаете.

Листинг 9.6. Компонент CreatePost (src/components/post/Create.js)

```
import PropTypes from 'prop-types';
import React from 'react';
import Filter from 'bad-words';
import classnames from 'classnames';
import DisplayMap from '../map/DisplayMap';
import LocationTypeAhead from '../map/LocationTypeAhead';
class CreatePost extends React.Component {
  static propTypes = {
    onSubmit: PropTypes.func.isRequired
  };
  constructor(props) {
    super(props);
    this.initialState = {
      content: '',
      valid: false,
      showLocationPicker: false,
      location: {
        lat: 34.1535641,
        lng: -118.1428115,
        name: null
      },
      locationSelected: false
    };
    this.state = this.initialState;
    this.filter = new Filter();
    this.handlePostChange = this.handlePostChange.bind(this);
    this.handleRemoveLocation = this.handleRemoveLocation.bind(this);
    this.handleSubmit = this.handleSubmit.bind(this);
    this.handleToggleLocation = this.handleToggleLocation.bind(this);
```

```
  this.onLocationSelect = this.onLocationSelect.bind(this);
  this.onLocationUpdate = this.onLocationUpdate.bind(this);
  this.renderLocationControls = this.renderLocationControls.bind(this);
}
handlePostChange(event) {
  const content = this.filter.clean(event.target.value);
  this.setState(() => {
    return {
      content,
      valid: content.length <= 300
    };
  });
}
handleRemoveLocation() {
  this.setState(() => ({
    locationSelected: false,
    location: this.initialState.location
  }));
}
handleSubmit(event) {
  event.preventDefault();
  if (!this.state.valid) {
    return;
  }
  const newPost = {
    content: this.state.content
  };
  if (this.state.locationSelected) {
    newPost.location = this.state.location;
  }
  this.props.onSubmit(newPost);
  this.setState(() => ({
    content: '',
    valid: false,
    showLocationPicker: false,
    location: this.defaultLocation,
    locationSelected: false
  }));
}
onLocationUpdate(location) {
  this.setState(() => ({ location }));
}
onLocationSelect(location) {
  this.setState(() => ({
    location,
    showLocationPicker: false,
    locationSelected: true
  }));
}
handleToggleLocation(event) {
  event.preventDefault();
```

```
    this.setState(state => ({ showLocationPicker:
    !state.showLocationPicker }));
  }
  renderLocationControls() {
    return (
      <div className="controls">
        <button onClick={this.handleSubmit}>Post</button>
        {this.state.location && this.state.locationSelected ? (
          <button onClick={this.handleRemoveLocation}
            className="open location-indicator">
            <i className="fa-location-arrow fa" />
            <small>{this.state.location.name}</small>
          </button>
        ) : (
          <button onClick={this.handleToggleLocation}
            className="open">
            {this.state.showLocationPicker ? 'Cancel' : 'Add
             location'}{' '}
            <i className={classnames('fa', {
              'fa-map-o': !this.state.showLocationPicker,
              'fa-times': this.state.showLocationPicker
            })}
          />
          </button>
        )}
      </div>
    );
  }
  render() {
    return (
      <div className="create-post">
        <textarea
          value={this.state.content}
          onChange={this.handlePostChange}
          placeholder="What's on your mind?"
        />
        {this.renderLocationControls()}
        <div
          className="location-picker"
          style={{ display: this.state.showLocationPicker ? 'block'
          : 'none' }}
        >
          {!this.state.locationSelected && (
            <LocationTypeAhead
              onLocationSelect={this.onLocationSelect}
              onLocationUpdate={this.onLocationUpdate}
            />
          )}
          <DisplayMap
            displayOnly={false}
```

```
            location={this.state.location}
            onLocationSelect={this.onLocationSelect}
            onLocationUpdate={this.onLocationUpdate}
          />
        </div>
      </div>
    );
  }
}

export default CreatePost;
```

Это немного более сложный компонент, чем вы создавали в предыдущих главах. С его помощью можете формировать сообщения и добавлять к ним местоположение. По моему опыту, тестирование более крупных и более сложных компонентов еще лучше подчеркивает важность ясных, читаемых тестов. Если вы не способны прочитать или объяснить свой тестовый файл, что вы или другой разработчик станете делать в будущем?

В листинге 9.7 показан предлагаемый каркас тестов для компонента `CreatePost`. У вас недостаточно методов, чтобы затруднить чтение тестов, но, если у компонента их больше, добавьте вложенные блоки `describe`, чтобы упростить понимание. Функции в листинге 9.7 будут выполняться исполнителем тестов (в данном случае Jest), и в рамках этих тестов можете сделать свои утверждения. Большинство тестов придерживаются одного и того же шаблона. Вы импортируете тестируемый код, устраняете любые зависимости, чтобы изолировать тесты до одной единицы функциональности (так что это модульные тесты), а затем исполнитель тестов и библиотека утверждений будут работать вместе для запуска тестов.

Листинг 9.7. Тестирование компонента CreatePost (src/components/post/Create.test.js)

```
jest.mock('mapbox');
import React from 'react';
import renderer from 'react-test-renderer';

import CreatePost from '../../../src/components/post/Create';

describe('CreatePost', () => {
  test('snapshot', () => {

  });
  test('handlePostChange', () => {

  });
  test('handleRemoveLocation', () => {

  });
  test('handleSubmit', () => {

  });
```

Здесь используется один вызов describe, но в больших тестовых файлах их может быть много и их даже можно вкладывать

Создание теста для всех методов компонента, включая моментальный снимок, чтобы убедиться в правильности рендеринга

```
  test('onLocationUpdate', () => {

  });
  test('handleToggleLocation', () => {

  });
  test('onLocationSelect', () => {

  });
  test('renderLocationControls', () => {

  });
});
```

Если вы будете придерживаться последовательной схемы рассмотрения всех частей компонента, который должен быть протестирован, то разработка и тестирование станут более тщательными. Не стесняйтесь использовать какую-либо структуру, имеющую для вас наибольший смысл, — как та, что была полезна для меня и команд, в которых я работал. Я также счел полезным начать создавать тесты, написав разные блоки `describe` и `test` для компонента или модуля, прежде чем разрабатывать какие-либо другие тесты. Я обнаружил, что мне легче продумать варианты (с ошибкой, без ошибок, с условием и т. д.), если делаю все сразу.

Как насчет других типов тестирования?

Возможно, вам интересно, как тестировать потоки пользователей, выполнять кроссбраузерное и другие типы тестирования, которые я здесь не рассматриваю. Специализированными формами тестирования, как правило, занимаются разработчик или особая команда программистов. Команды QA и SET (разработчики программного обеспечения теста) обычно имеют множество специализированных инструментов, которые позволяют им взять приложение и симулировать все существующие сложные потоки.

Эти типы тестирования (интеграционное тестирование) могут включать взаимодействие одной или нескольких различных систем. Вспомните тестовую пирамиду на рис. 9.1 — написание этих тестов может занять много времени, их трудно поддерживать и, как правило, они дорого стоят. Когда вы думаете о тестировании интерфейсных приложений, то, вероятно, предположите, что будут задействованы подобные тесты. Мы видели, что это не так (большинство тестов, которые пишут разработчики, не являющиеся разработчиками QA, — это тесты модулей или низкоуровневые интеграционные тесты). Если вам интересно подробнее рассмотреть эти инструменты, вот несколько примеров, которые вы можете использовать в качестве трамплина, чтобы больше узнать о тестировании высокого уровня:

- *Selenium* — www.seleniumhq.org;
- *Puppeteer* — https://github.com/GoogleChrome/puppeteer;
- *Protractor* — www.protractortest.org/#/.

С помощью установленного каркаса можно протестировать компонент `CreatePost`, начиная с конструктора. Помните, что в конструкторе устанавливается начальное состояние, получают привязку методы класса и могут выполняться другие настройки. Чтобы протестировать эту часть компонента `CreatePost`, нужно задействовать еще один инструмент, о котором я упоминал ранее, — Sinon. Вам нужны какие-то тестовые функции, которые вы можете предоставить своему компоненту для использования и которые не зависят от других модулей. С помощью Jest можно создавать mock-функции для теста, которые помогают сфокусировать тесты на самом компоненте и не испытывать весь код целиком. Помните, я говорил, что тесты должны прекращать работать, когда вы меняете свой код? Это так, но в то же время изменение одного теста не должно отменять другие. Как и в случае с обычным кодом, ваши тесты не должны быть связаны и обязаны заботиться только о фрагменте кода, который тестируют.

mock-функции Jest помогают нам не только изолировать код, но и делать больше утверждений. Вы можете сделать утверждения о том, как компонент использовал mock-функцию: была ли она вызвана, с какими аргументами и т. д. В листинге 9.8 показана настройка теста моментального снимка для компонента и имитируются некоторые базовые свойства компонентов с помощью Jest.

Листинг 9.8. Ваш первый тест (src/components/post/Create.test.js)

```
jest.mock('mapbox');
import React from 'react';
import renderer from 'react-test-renderer';

import CreatePost from '../../../src/components/post/Create';

describe('CreatePost', () => {
  test('snapshot', () => {
    const props = { onSubmit: jest.fn() };
    const component = renderer.create(<CreatePost {...props} />);
    const tree = component.toJSON();
    expect(tree).toMatchSnapshot();
  });
  //...
});
```

Использование функции jest.mock, чтобы приказать Jest применять имитацию вместо модуля при выполнении тестов

Создание блока test внутри внешнего блока describe, выполненного ранее

Применение средства React test renderer для создания компонента и передачи свойств

Подтверждение соответствия моментального снимка

Вызов метода jSON для создания моментального снимка

Создание mock-свойств и использование Jest для разработки mock-функции

Теперь, когда один тест готов, можете протестировать некоторые другие характеристики компонента. Компонент несет ответственность в основном за то, чтобы пользователи могли создавать сообщения и прикреплять к ним информацию о местоположении, поэтому нужно протестировать эти области функциональности. Начните с тестирования создания сообщений. В листинге 9.9 показано, как тестировать методы создания сообщений в компоненте.

Листинг 9.9. Тестирование создания сообщений (src/components/post/Create.test.js)

```
jest.mock('mapbox');
import React from 'react';
import renderer from 'react-test-renderer';

import CreatePost from '../../../src/components/post/Create';

describe('CreatePost', () => {
  test('snapshot', () => {
    const props = { onSubmit: jest.fn() };
    const component = renderer.create(<CreatePost {...props} />);
    const tree = component.toJSON();
    expect(tree).toMatchSnapshot();
  });
  test('handlePostChange', () => {
    const props = { onSubmit: jest.fn() };
    const mockEvent = { target: { value: 'value' } };
    CreatePost.prototype.setState = jest.fn(function(updater) {
      this.state = Object.assign(this.state, updater(this.state));
    });

    const component = new CreatePost(props);
    component.handlePostChange(mockEvent);
    expect(component.setState).toHaveBeenCalled();
    expect(component.setState.mock.calls.length).toEqual(1);
    expect(component.state).toEqual({
      valid: true,
      content: mockEvent.target.value,
      location: {
        lat: 34.1535641,
        lng: -118.1428115,
        name: null
      },
      locationSelected: false,
      showLocationPicker: false
    });

  });
  test('handleSubmit', () => {
    const props = { onSubmit: jest.fn() };
    const mockEvent = {
      target: { value: 'value' },
      preventDefault: jest.fn()
    };
    CreatePost.prototype.setState = jest.fn(function(updater) {
```

mock-функция setState, чтобы вы могли убедиться, что компонент вызывает ее и обновление сообщения приводит к правильному обновлению состояния

Создание набора mock-свойств

Непосредственное создание экземпляра компонента и вызов его методов

Убедитесь, что компонент вызывает правильные методы и метод правильно обновил состояние

Создание еще одного mock-события для имитации того, что компонент получит от события

Еще одна mock-функция setState

```
      this.state = Object.assign(this.state, updater(this.state));
    });

    const component = new CreatePost(props);
    component.setState(() => ({
      valid: true,
      content: 'cool stuff!'
    }));
    component.state = {
      valid: true,
      content: 'content',
      location: 'place',
      locationSelected: true
    };
    component.handleSubmit(mockEvent);
    expect(component.setState).toHaveBeenCalled();
    expect(props.onSubmit).toHaveBeenCalledWith({
      content: 'content',
      location: 'place'
    });
  });
});
```

Создание экземпляра другого компонента и установка его состояния для имитации ввода пользователем содержимого сообщения

Непосредственное изменение состояния компонента (в целях тестирования)

Обработка отсылки сообщения с mock-событием, которое вы создали, и утверждение, что имитации были вызваны

Наконец, нужно проверить оставшуюся часть функциональности компонента. Помимо того что пользователи могут создавать сообщения, компонент `CreatePost` обрабатывает выбор местоположения пользователем. Другие компоненты обрабатывают обновление местоположения через обратные вызовы, переданные как свойства, но все равно следует протестировать методы компонента `CreatePost`, связанные с этой функцией.

Помните, что вы реализовали метод подрендеринга в `CreatePost`, который использовали для упрощения чтения вывода метода `render` из `CreatePost` и уменьшения беспорядка. Можете протестировать это так же, как тестировали компоненты с помощью Enzyme или React test renderer. В листинге 9.10 показаны остальные тесты для компонента `CreatePost`.

Листинг 9.10. Тестирование создания сообщения (src/components/post/Create.test.js)

```
jest.mock('mapbox');
import React from 'react';
import renderer from 'react-test-renderer';

import CreatePost from '../../../src/components/post/Create';

describe('CreatePost', () => {
  test('handleRemoveLocation', () => {
    const props = { onSubmit: jest.fn() };
    CreatePost.prototype.setState = jest.fn(function(updater) {
      this.state = Object.assign(this.state, updater(this.state));
    });
    const component = new CreatePost(props);
    component.handleRemoveLocation();
```

mock-функция setState

Включение функции handleRemoveLocation

```
  expect(component.state.locationSelected).toEqual(false);
});
test('onLocationUpdate', () => {
  const props = { onSubmit: jest.fn() };
  CreatePost.prototype.setState = jest.fn(function(updater) {
    this.state = Object.assign(this.state, updater(this.state));
  });
  const component = new CreatePost(props);
  component.onLocationUpdate({
    lat: 1,
    lng: 2,
    name: 'name'
  });
  expect(component.setState).toHaveBeenCalled();
  expect(component.state.location).toEqual({
    lat: 1,
    lng: 2,
    name: 'name'
  });
});
test('handleToggleLocation', () => {
  const props = { onSubmit: jest.fn() };
  const mockEvent = {
    preventDefault: jest.fn()
  };
  CreatePost.prototype.setState = jest.fn(function(updater) {
    this.state = Object.assign(this.state, updater(this.state));
  });
  const component = new CreatePost(props);
  component.handleToggleLocation(mockEvent);
  expect(mockEvent.preventDefault).toHaveBeenCalled();
  expect(component.state.showLocationPicker).toEqual(true);
});
test('onLocationSelect', () => {
  const props = { onSubmit: jest.fn() };
  CreatePost.prototype.setState = jest.fn(function(updater) {
    this.state = Object.assign(this.state, updater(this.state));
  });
  const component = new CreatePost(props);
  component.onLocationSelect({
    lat: 1,
    lng: 2,
    name: 'name'
  });

test('onLocationSelect', () => {
  const props = { onSubmit: jest.fn() };
  CreatePost.prototype.setState = jest.fn(function(updater) {
    this.state = Object.assign(this.state, updater(this.state));
  });
  const component = new CreatePost(props);
  component.onLocationSelect({
    lat: 1,
    lng: 2,
```

Убедитесь, что состояние обновлено правильно

Повторите этот процесс для остальных методов компонента

```
      name: 'name'
    });
    expect(component.setState).toHaveBeenCalled();
    expect(component.state.location).toEqual({
      lat: 1,
      lng: 2,
      name: 'name'
    });
  });

  test('renderLocationControls', () => {
    const props = { onSubmit: jest.fn() };
    const component = renderer.create(<CreatePost {...props} />);
    let tree = component.toJSON();
    expect(tree).toMatchSnapshot();
  });
});
```

Создание еще одного теста моментального снимка для метода подрендеринга

9.3.4. Покрытие тестированием

Теперь, когда вы приобрели опыт проверки некоторых компонентов, посмотрим на охват тестирования и достигнутый прогресс. В терминале остановите исполнитель тестов и выполните команду, показанную в листинге 9.11. Эта команда содержит опцию покрытия, включенную в Jest.

Листинг 9.11. Обеспечение охвата тестированием (корень проекта)

```
> npm run test:w
```

Как только исполнитель тестов закончит тестирование, он должен выдать цветную таблицу, которая должна выглядеть примерно как рис. 9.3 (с меньшим охватом). Здесь показана информация об охвате Jest с аннотациями для каждого столбца. Существуют различные формы читабельных отчетов о покрытии кода (например, HTML), но вывод терминала наиболее полезен во время разработки, поскольку обеспечивает немедленную обратную связь.

Istanbul — инструмент, генерирующий статистику, приведенную на рис. 9.3. Если хотите получить более подробную информацию о покрытии, откройте каталог покрытия, который должен был быть создан командой `jest`, включающей опцию покрытия. В этом каталоге Istanbul должен был сформировать несколько файлов. Если вы откроете файл `./coverage/lcov-report/index.html` в браузере, то должны увидеть что-то вроде рис. 9.4.

Вывод Istanbul полезен, но вы также можете подробно проанализировать разные файлы и получить более подробную информацию об отдельных файлах. Каждый файл должен отображать информацию о том, сколько раз были покрыты разные строки, а какие — не покрыты вообще. В большинстве случаев сводка на верхнем уровне довольно хороша, но иногда вам требуется просмотреть отдельные отчеты, например, как на рис. 9.5. Как только при написании тестов я рассмотрю все случаи, то по крайней мере один раз проглядываю эти файлы, чтобы убедиться, что не пропущены никакие граничные случаи или логические ветви.

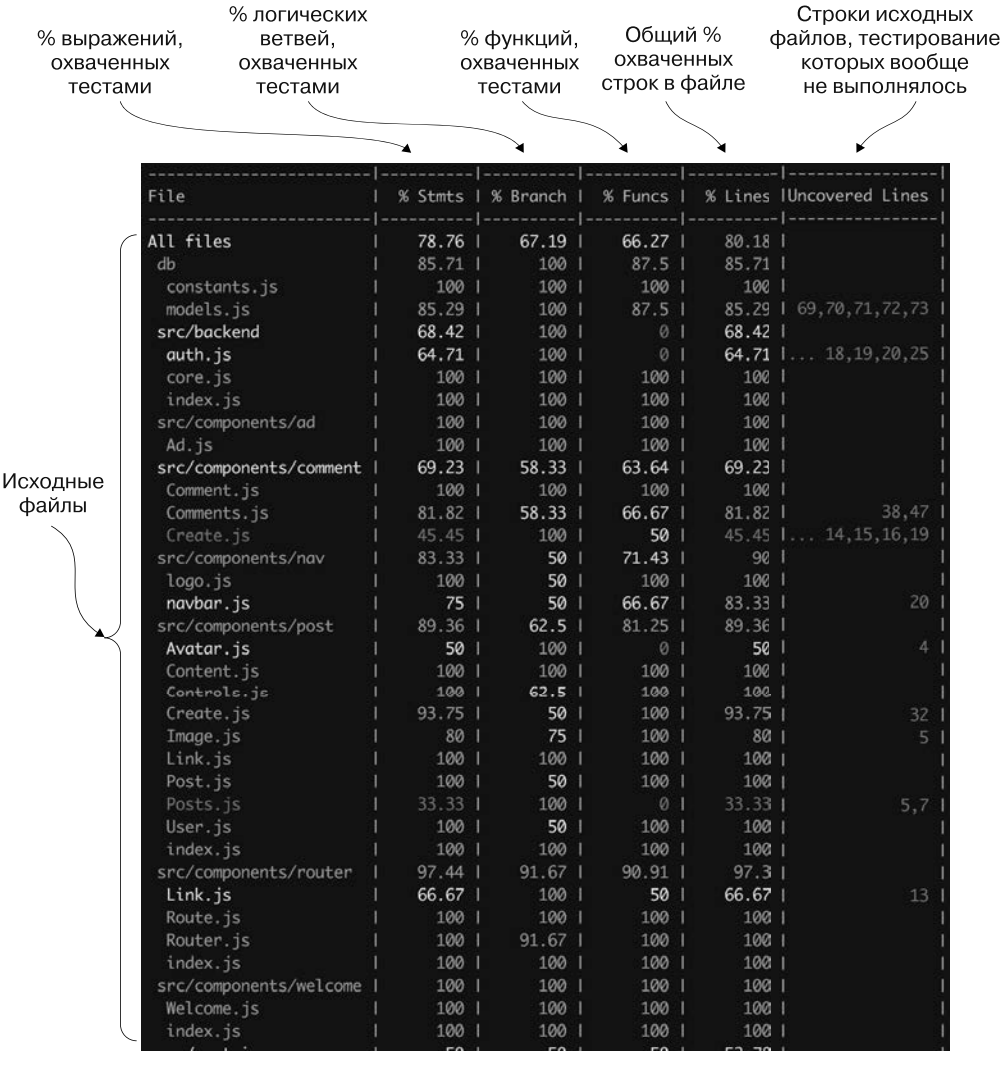
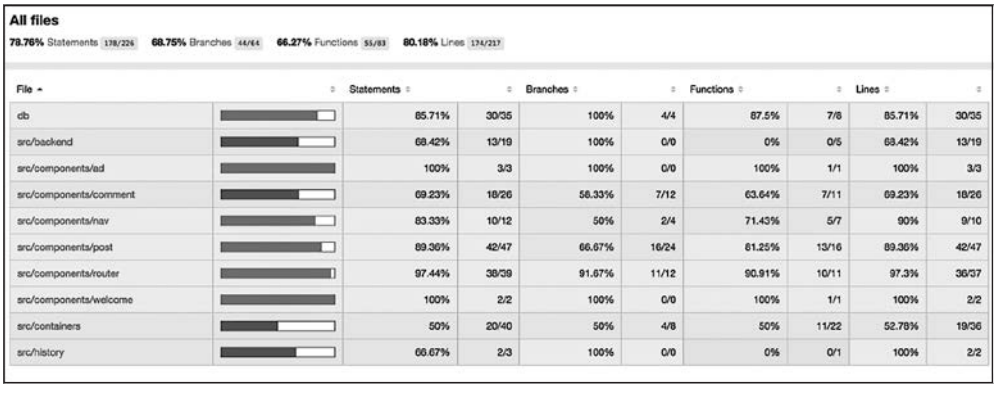


Рис. 9.3. Результаты тестирования покрытия от Jest показывают статистику охвата различных файлов в вашем проекте. Каждый столбец отражает определенный показатель покрытия. Для каждого типа покрытия Jest приводит охват в процентах. Выражения и функции — это просто инструкции и функции JavaScript, а ветви — логические ветви. Если в вашем тесте не рассматривается часть инструкции if, это должно отразиться на охвате кода как в столбце непокрытых строк, так и в процентах статистики для ветви

Покрытие тестированием — важный и полезный инструмент для разработки программного обеспечения, но не считайте его волшебной гарантией того, что код работает. Вы можете получить 100%-ный охват и все еще иметь код, который прекращает работу. И наоборот, технически возможен код, который работает при охвате 0 %. *Охват* заключается в том, чтобы убедиться, что тесты проходят все фрагменты

кода. Это не гарантирует отсутствия ошибок или высокую производительность, но должно рассматриваться как важный источник сведений при определении того, насколько код завершен. Я работал в командах, где наше определение успеха для конкретной истории пользователя или задачи включало среди прочего покрытие кода выше 80 % и не уменьшало охват в целом. Используйте покрытие в качестве ориентира для определения тех частей кода, которые были или не были протестированы, и для проверки прогресса тестирования.

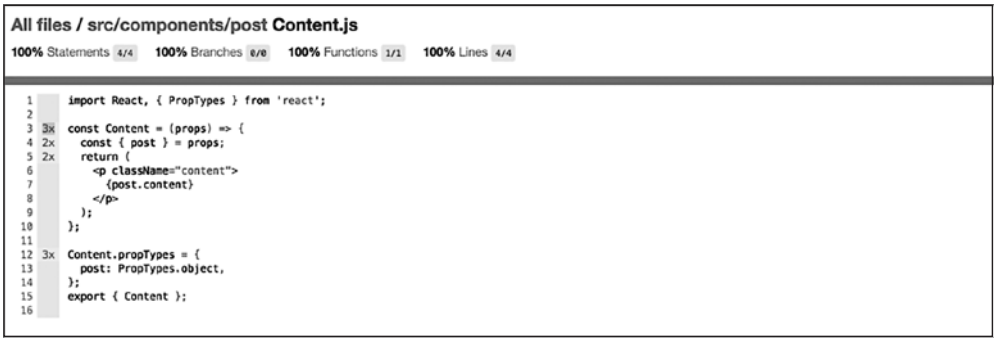


All files

78.76% Statements 178/226 **68.75%** Branches 44/64 **66.27%** Functions 55/83 **80.18%** Lines 174/217

File		Statements		Branches		Functions		Lines	
db		85.71%	30/35	100%	4/4	87.5%	7/8	85.71%	30/35
src/backend		68.42%	13/19	100%	0/0	0%	0/5	68.42%	13/19
src/components/ad		100%	3/3	100%	0/0	100%	1/1	100%	3/3
src/components/comment		69.23%	18/26	58.33%	7/12	63.64%	7/11	69.23%	18/26
src/components/nav		83.33%	10/12	50%	2/4	71.43%	5/7	90%	9/10
src/components/post		89.36%	42/47	66.67%	16/24	81.25%	13/16	89.36%	42/47
src/components/router		97.44%	38/39	91.67%	11/12	90.91%	10/11	97.3%	36/37
src/components/welcome		100%	2/2	100%	0/0	100%	1/1	100%	2/2
src/containers		50%	20/40	50%	4/8	50%	11/22	52.78%	19/36
src/history		66.67%	2/3	100%	0/0	0%	0/1	100%	2/2

Рис. 9.4. Istanbul генерирует метаданные покрытия в машиночитаемых и удобочитаемых форматах. Показанный здесь отчет о покрытии полезен для более детального изучения покрытия кода. Вы даже можете сортировать по разным столбцам и приоритизировать файлы с малым охватом. Обратите внимание на то, что существуют столбцы для утверждений, ветвей (if/else statements), функций (какие из них были вызваны) и строк (строк кода)



All files / src/components/post Content.js

100% Statements 4/4 **100%** Branches 0/0 **100%** Functions 1/1 **100%** Lines 4/4

```
1    import React, { PropTypes } from 'react';
2
3  3x const Content = (props) => {
4  2x   const { post } = props;
5  2x   return (
6         <p className="content">
7           {post.content}
8         </p>
9       );
10   };
11
12 3x Content.propTypes = {
13     post: PropTypes.object,
14   };
15   export { Content };
16
```

Рис. 9.5. Отчет о покрытии отдельных файлов, созданный Istanbul. Вы можете увидеть, сколько раз разные строки были или не были покрыты, и понять, какие именно части кода были покрыты

Упражнение 9.2. Рассмотрение покрытия

В этой главе мы говорили о покрытии тестированием. Означает ли 100%-ный охват кода тестированием, что он совершенен? Какую роль в тестировании играет покрытие кода?

9.4. Резюме

Из этой главы вы узнали о некоторых принципах, на которых базируется тестирование, и способах тестирования React-приложений.

- *Тестирование* — это процесс проверки предположений о программном обеспечении. Оно помогает вам лучше планировать компоненты, предотвратить сбои в будущем и повысить уверенность в коде. А также играет важную роль в процессе быстрой разработки.
- Ручное тестирование не очень хорошо масштабируется, потому что никакое количество людей не может быстро или правильно протестировать сложное программное обеспечение.
- В процессе тестирования программного обеспечения мы используем различные инструменты, начиная с тех, которые запускают тесты, и заканчивая определяющими, какое количество кода покрывается тестированием.
- Различные типы тестов следует задействовать в разных пропорциях. *Модульные* тесты должны быть наиболее распространенными и простыми, дешевыми и быстро создаваемыми. *Интеграционные* тесты проверяют взаимодействие многих различных частей системы, они могут быть нестабильными, и для их написания требуется больше времени. Их не должно быть много.
- Вы можете протестировать компоненты React с помощью различных инструментов. Поскольку это просто функции, можно лишь проверить их как таковые. Такие инструменты, как Enzyme, упрощают тестирование компонентов React.
- Ясные тесты, как и любой ясный код, легко читаются, хорошо организованы и используют модульные, сервисные и интеграционные тесты в соответствующих пропорциях. Они должны обеспечить вам уверенность в том, что все функционирует определенным образом, и гарантировать, что изменения в компоненте могут быть учтены.

В следующей главе рассмотрим более надежную реализацию приложения Letters Social и изучим архитектурный шаблон Redux. Прежде чем читать дальше, посмотрите, можете ли вы продолжать оттачивать свои навыки тестирования и получить для приложения тестовое покрытие выше 90 %!

Часть III
Архитектура React-приложений

К концу части II пример приложения Letters Social из каркаса статической страницы должен превратиться в динамический пользовательский интерфейс с маршрутизацией, аутентификацией и динамическими данными. В части III вы разовьете то, что создали, изучив некоторые дополнительные темы в React.

В главах 10 и 11 вы изучите архитектуру приложения Flux и реализуете Redux. Redux — это разновидность шаблона Flux, который де-факто стал решением для управления состоянием больших React-приложений. Вы изучите концепции Redux и изменение React-приложения для использования Redux в качестве решения для управления состоянием. По мере выполнения всего этого вы продолжите добавлять Letters Social новые свойства, включая комментарии и возможность отмечать сообщения как понравившиеся.

В главе 12 мы сделаем еще один шаг и рассмотрим, как React работает на сервере. Благодаря доступности серверной среды выполнения node.js вы сможете выполнить код React на сервере. Вы исследуете рендеринг на стороне сервера с помощью React и интегрируете управление состоянием Redux в процесс. А также интегрируете `React Router` — популярную библиотеку маршрутизации для React.

Наконец, в главе 13 вы немного отойдете от React для Интернета и изучите React Native. React Native — это еще один проект React, позволяющий писать React-приложения, которые можно запускать на мобильных устройствах iOS и Android.

К концу части III будет создано целое приложение, в полной мере использующее рендеринг React, Redux и на стороне сервера. Так вы завершите знакомство с React, но сможете продолжить работать с ней и изучить другие продвинутые темы, такие как React Native.

10 Архитектура приложения Redux

- Действия Redux, хранилища, редукторы и промежуточное ПО.
- Тестирование действий, хранилищ, редукторов и промежуточного программного обеспечения Redux.

К этому моменту вы можете создавать React-приложения, которые протестированы, поддерживают динамические данные, принимают входные данные и взаимодействуют с удаленными API. Все это — бóльшая часть действий обычного веб-приложения; единственное, что осталось, — это практика. Применение полученных навыков поможет вам справиться с React, но существует еще одна важная область, которую нужно охватить для создания более крупных и сложных приложений, — архитектуры приложений. *Архитектура приложения* — это «процесс определения структурированного решения, отвечающего всем техническим и эксплуатационным требованиям при оптимизации общих атрибутов качества, таких как производительность, безопасность и управляемость» («Руководство по архитектуре приложений Microsoft», второе издание). Архитектура спрашивает: «Хорошо, мы можем это реализовать, но как сделать лучше и более последовательно?» — о том, как и насколько хорошо организовано приложение, как перемещаются данные, а ответственность делегируется различным частям системы.

Каждое приложение характеризуется неявной архитектурой особого рода просто потому, что имеет собственную структуру и выполняет свои функции определенным образом. То, о чем я говорю, — это стратегии и парадигмы для построения сложных приложений. React допускает ошибку, так как использует минимальный или неконсервативный фреймворк, ориентированный на пользовательский интерфейс, поэтому не имеет подходящей встроенной стратегии, поскольку вы создаете более сложные приложения.

Только то, что нет встроенной стратегии, которую вы могли бы задействовать, не означает, что там нет опций. Существует множество подходов к созданию сложных приложений с React, и многие из них основаны на модели Flux, популяризированной разработчиками в Facebook. Flux отличается от популярной архитектуры MVC приверженностью к однонаправленным потокам данных, внедрением новых концепций (диспетчеров, действий, хранилищ) и т. п. Flux и MVC связаны с моментами, которые «выше» внешнего вида приложения и даже некоторых из конкретных

библиотек и технологий, с применением которых оно построено. Они относятся скорее к организации приложения, к тому, как перемещаются данные и ответственность делегируется различным частям системы.

В этой главе рассматривается один из наиболее широко используемых и хорошо известных вариантов шаблона Flux — Redux. Совершенно обычно то, что Redux используется с приложениями React, но на самом деле его можно применять с большинством фреймворков JavaScript (как изнутри, так и другим способом). В этой и следующей главах описываются основные концепции Redux — действия, промежуточное программное обеспечение, редукторы, хранилище и т. д., а также интеграция Redux с вашим приложением React. *Действия* в Redux представляют собой выполненную работу — получение пользовательских данных, ведение журнала пользователя и т. д. *Редукторы* определяют, как должно измениться состояние, в *хранилище* содержится централизованная копия состояния, а *промежуточное программное обеспечение* позволяет пользователю воздействовать на процесс.

Получение исходного кода

Как и прежде, вы можете получить исходный код примеров из этой главы, перейдя в репозиторий GitHub по адресу github.com/react-in-action/letters-social. Если планируете начать работу здесь самостоятельно с нуля, возьмите исходный код примеров из главы 9 (если изучили ее и сами выполнили примеры) или обратитесь к ветви, относящейся к данной главе (chapter-10-11).

Помните, что каждая ветвь содержит итоговый код главы (например, chapter-10-11 содержит код, получаемый в конце глав 10 и 11). Вы можете в оболочке командной строки выполнить одну из следующих команд по своему выбору, чтобы получить код примеров из текущей главы. Если репозитория вообще нет, выполните команду:

```
git clone git@github.com:react-in-action/letters-social.git
```

Если у вас уже клонирован репозиторий, то следующую:

```
git checkout chapter-10-11
```

Возможно, вы перешли сюда из другой главы, поэтому стоит проверить, установлены ли у вас все нужные зависимости, с помощью команды:

```
npm install
```

10.1. Архитектура приложения Flux

Современные приложения более функциональны, чем существовавшие ранее, и, соответственно, более сложны — как изнутри, так и снаружи. Разработчики уже давно знают о беспорядке, который может получиться из сложного приложения, развивающегося без согласованных шаблонов проектирования. Со спагетти-кодом не только скучно работать — он замедляет деятельность разработчиков и, следова-

тельно, бизнеса. Помните, как в последний раз вам приходилось работать в большой базе кода, полной одноразовых решений и плагинов jQuery? Наверное, это было невесело. Для борьбы с дезорганизацией разработчики предложили такие парадигмы, как MVC (Model — View — Controller, «Модель — Представление — Контроллер»), чтобы организовать функциональность разработки приложения и руководства процессом. Flux (и расширение Redux) — это тоже попытка помочь вам справиться с повышенной сложностью приложения.

- *Модель* — данные для приложения. Обычно это существительное, например User, Account или Post. Ваша модель должна иметь по крайней мере базовые методы для управления соответствующими данными. В более абстрактном смысле модель представляет собой необработанные данные или знания. Здесь данные пересекаются с кодом приложения. Например, база данных способна хранить несколько свойств, таких как `accessScopes`, `authenticated` и т. д. Но модель сможет использовать эти данные для такого метода, как `isAllowedAccessForResource()`, который воздействует на базовые данные для модели. Модель — это место, где необработанные данные сходятся с кодом приложения.
- *Представление* — представление вашей модели. Часто оно является интерфейсом пользователя. В представлении не должно быть никакой логики, не связанной с представлением данных. Для интерфейсных фреймворков это обычно означает, что конкретное представление напрямую связано с ресурсом и реализует для него действия CRUD (создавать, читать, обновлять, удалять). Интерфейсные приложения так больше не выполняются.
- *Контроллер* — это клей, который связывает модель и отображение. Контроллеры, как правило, должны быть только клеем и немногим больше (например, у них не должно быть сложного представления или логики базы данных). Обычно стоит ожидать, что контроллеры будут обладать гораздо меньшей способностью изменять данные, чем модели, с которыми они взаимодействуют.

Парадигмы, на которых мы собираемся сосредоточиться в этой главе (Flux и Redux), отходят от этих концепций, но все еще призваны помочь вам создать масштабируемую, разумную и эффективную архитектуру приложений.

Redux обязан своим происхождением и дизайном шаблону, популяризированному в Facebook, который называется *Flux*. Тех, кто знаком с широко распространенным шаблоном MVC, который использует Ruby on Rails и другие фреймворки приложений, хочу предупредить: Flux отличается от того, к чему вы привыкли. Вместо разбиения приложения на модели, отображения и контроллеры Flux определяет для себя несколько других частей:

- *хранилище* — хранилища содержат состояние приложения и логику, они чем-то похожи на модель в традиционном MVC. Однако вместо представления одной записи базы данных они управляют состоянием многих объектов. В отличие от модели, вы представляете данные, когда это имеет смысл, и не ограничены ресурсами;

- *действия* — вместо непосредственного обновления состояния приложения Flux изменяют свое состояние, создавая действия, которые делают это;
- *представление* — пользовательский интерфейс, как правило, React, хотя Flux она не требуется;
- *диспетчер* — центральный координатор действий и обновлений хранилищ.

На рис. 10.1 дана общая схема Flux.

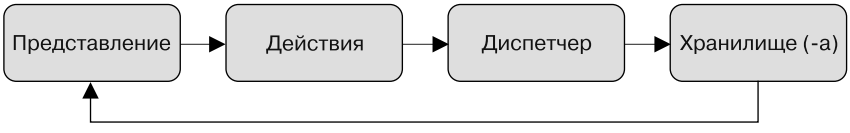


Рис. 10.1. Схема Flux

В шаблоне Flux (см. рис. 10.1) действия создаются из представлений — это может быть пользователь, щелкающий на чем-то кнопкой мыши. В дальнейшем поступающие действия обрабатывает диспетчер. Затем они отправляются в соответствующее хранилище для обновления состояния. Состояние, изменившись, уведомляет представление о необходимости использования новых данных (если это применимо). Обратите внимание на отличия от типичного фреймворка в стиле MVC, где и представление, и модель (здесь, например, хранилище) смогут обновить друг друга. Этот двунаправленный поток данных отличается от однонаправленного потока, типичного для архитектур Flux. Также обратите внимание на то, что здесь отсутствует промежуточное программное обеспечение: его можно создать в Flux, однако здесь это не обязательно, как в Redux, поэтому мы его опускаем.

Возможно, некоторые из этих частей вам знакомы, но поток данных в Flux может не быть таким, как в приложениях в стиле MVC. Как уже упоминалось, потоки данных в парадигме Flux однонаправленны, в отличие от двунаправленных реализаций типа MVC. Обычно это означает, что нет единой точки, откуда исходят данные, многие части системы имеют право изменять состояние, а оно часто децентрализовано по всему приложению. Во многих случаях такой подход работает хорошо, но в крупных приложениях способен запутать при отладке и в ходе работы.

Подумайте, как это будет выглядеть в среднем или большом приложении. Скажем, у вас была коллекция моделей (пользователь, учетная запись и аутентификация), связанных с их собственными контроллерами и представлениями. В любом данном месте приложения может быть сложно определить точное местоположение состояния, так как оно распределено по частям приложения (информацию о пользователе можно обнаружить в любой из трех упомянутых моделей).

Это не обязательно будет проблемой для небольших приложений и даже может хорошо работать в более крупных, но, вероятно, все усложнится в нетривиальных

клиентских приложениях. Например, что происходит, когда вам нужно изменить сведения, которые должны знать об изменениях состояния, использующие модель в 50 местах и контроллеры в 60? Усугубляет ситуацию то, что в некоторых интерфейсных структурах представления иногда действуют как модели (поэтому состояние еще более децентрализованное). Что будет верным для ваших данных? Если они распределены по представлениям и множеству различных моделей и все это при довольно сложной конфигурации, отслеживать их в уме будет сложно. Это может сделать состояние приложения несогласованным, что вызовет ошибки приложения, так что это проблема не только для разработчиков — напрямую затронуты и конечные пользователи.

Одна из причин, почему это сложно, заключается в том, что люди вообще не думают об изменениях, которые происходят со временем. Чтобы понять, как непросто управлять всем этим, представьте себе шахматную доску. Это не так уж сложно, в голове могут возникнуть одно или несколько изображений доски. Но сможете ли вы отслеживать, что происходит на каждой после 20 ходов; 30; всей игры? Мы должны строить системы, которые легче понимать и использовать, потому что запоминать асинхронные изменения данных во времени трудно. Например, подумайте о вызове удаленного API и применении данных для обновления состояния приложения. Это просто для нескольких случаев, но что если вам нужно вызывать 50 разных конечных точек и отслеживать входящие ответы, пока пользователь работает с приложением и вносит изменения, которые могут сделать взаимодействие с API более тесным? Трудно мысленно выстроить их все в одну линию и предсказать, каков будет результат изменений.

Вы уже заметили некоторое сходство между React и Flux. Оба являются относительно новыми подходами к разработке пользовательских интерфейсов и направлены на улучшение ментальной модели, с которой трудится разработчик. В каждом из них изменения должны быть легко поняты, и вы должны иметь возможность создавать пользовательский интерфейс таким образом, чтобы он помогал вам вместо того, чтобы мешать.

Как выглядит Flux в коде? Это прежде всего парадигма, поэтому имеется множество доступных библиотек, которые реализуют основные идеи Flux. Все они немного отличаются друг от друга своей реализацией Flux. Redux также делает это, даже несмотря на то, что наиболее широко применяется ее особая ветвь от Flux. Другие библиотеки Flux включают Flummox, Fluxxor, Reflux, Fluxible, Lux, McFly и MartyJS (хотя вряд ли они используются так же широко, как Redux).

10.1.1. Знакомьтесь с Redux: вариант Flux

Возможно, наиболее широко используемой и известной библиотекой, реализующей идеи Flux, является Redux. Redux — это библиотека, которая придерживается идей Flux, но немного модифицирует их. В своей документации Redux описывается как

«предсказуемый контейнер состояний для приложений JavaScript». В конкретных терминах это означает, что это библиотека, которая реализует концепции и идеи Flux по-своему.

Дать точные определения того, чем является Flux, а чем — нет, здесь не важно, но я должен осветить некоторые существенные различия между парадигмами Flux и Redux.

- *Redux использует одно хранилище* — вместо того чтобы находить информацию о состоянии в нескольких хранилищах по всему приложению, приложения Redux хранят все в одном месте. В Flux у вас может быть много разных хранилищ. Redux отказывается от этого и применяет одно глобальное хранилище.
- *Redux вводит редукторы*. Редукторы — это более стабильный подход к изменениям. В Redux состояние изменяется предсказуемым и детерминированным образом: одна часть состояния за раз и только в одном месте (глобальном хранилище).
- *Redux вводит промежуточное программное обеспечение*. Поскольку действия и поток данных однонаправленны, вы можете добавить промежуточное программное обеспечение в свое приложение Redux и ввести пользовательское поведение по мере обновления данных.
- *Действия Redux отвязаны от хранилища*. Создатели действий ничего не отправляют в хранилище, вместо этого они возвращают объекты действий, которые использует центральный диспетчер.

Вам эта разница может показаться незначительной, и это нормально: ваша цель — узнать о Redux, а не выполнять упражнения «найди различия». На рис. 10.2 показан обзор архитектуры Redux. Вы углубитесь в каждый из разделов, изучите, как они работают, и разработаете архитектуру Redux для своего приложения.

Как вы видите на рис. 10.2, основными элементами архитектуры Redux являются действия, хранилище и редукторы. Redux использует единый объект централизованного состояния, который обновляется определенными детерминированными способами. Действие создается, когда вы хотите обновить состояние (обычно из-за события, такого как щелчок кнопкой мыши). Действие будет иметь тип, поддерживающий определенный редуктор. Редуктор, который обрабатывает данный тип действия, создаст копию текущего состояния, изменит его с применением данных из действия, а затем вернет новое состояние. Когда хранилище обновится, слои представления (в нашем случае React) смогут прослушивать обновления и отвечать соответствующим образом. Обратите внимание и на то, что на рисунке представления просто читают обновления из хранилища — они не заботятся о том, как им передаются данные. Библиотека `react-redux` поддерживает передачу новых свойств компонентам при изменении хранилища, но представления все же только получают и отображают данные.

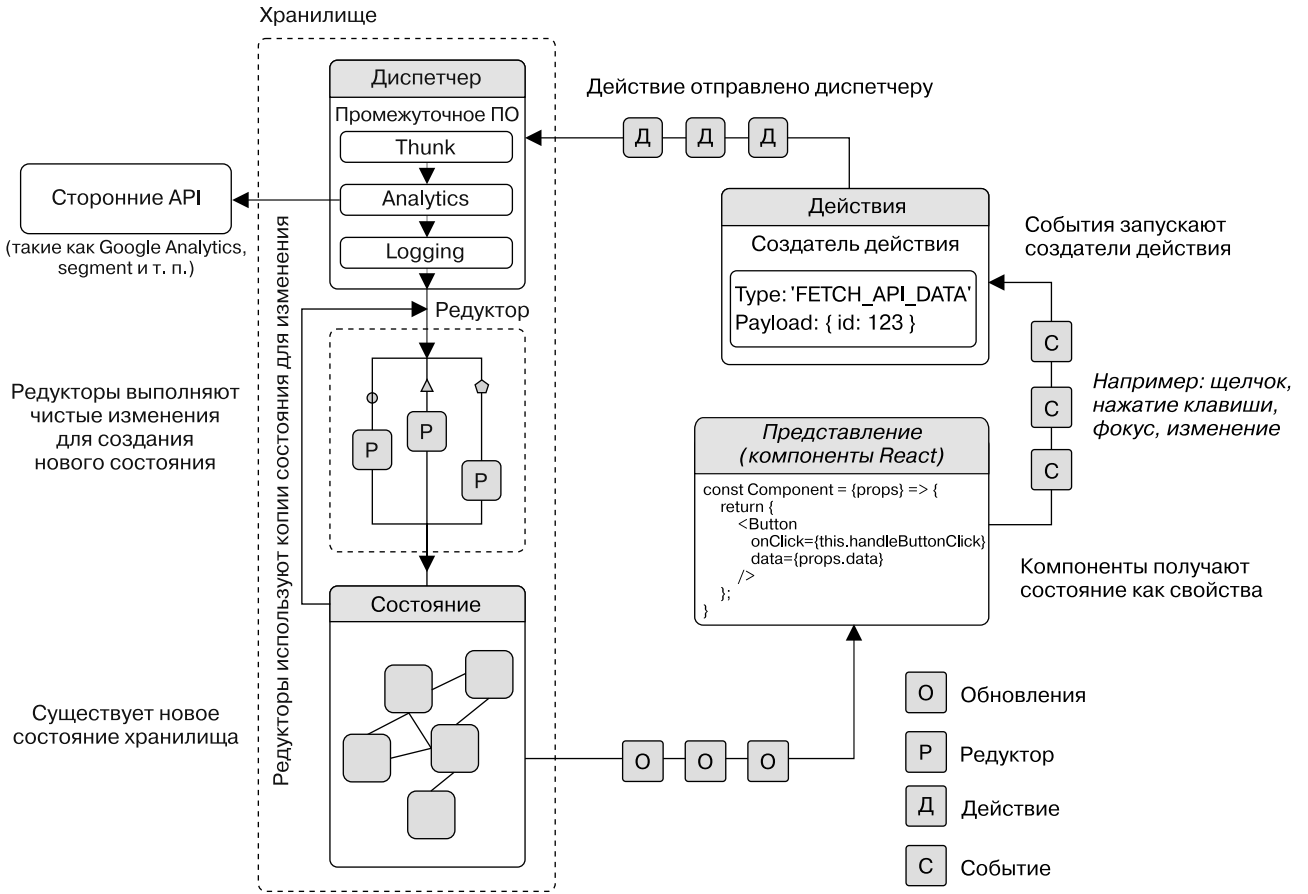


Рис. 10.2. Обзор Redux

10.1.2. Настройка для Redux

Redux — это парадигма для архитектуры вашего приложения, но это и библиотека, которую вы можете установить. Это одна из областей, где Redux доминирует над сырой реализацией Flux. Существует много реализаций парадигмы Flux — Flummox, Fluxxor, Reflux, Fluxible, Lux, McFly, MartyJS и др., и все они имеют разную степень поддержки сообщества и различные API. Redux пользуется значительной поддержкой сообщества, а библиотека Redux имеет небольшой мощный API, который помог ей стать одной из самых популярных и надежных библиотек для архитектуры React-приложения. Фактически Redux, использующийся с React, встречается так часто, что основные команды разработчиков каждой из библиотек регулярно взаимодействуют друг с другом и обеспечивают совместимость и узнаваемость функций. Некоторые даже работают в обеих командах, поэтому в целом между проектами существуют отличные согласие и взаимодействие.

Чтобы настроить Redux, сделайте следующее.

- Убедитесь, что выполнили команду `npm install` с исходным кодом из текущей главы, так что все правильные зависимости установлены локально. В этой главе вы начнете использовать некоторые новые библиотеки, в том числе `js-cookie`, `redux-mock-store` и `redux`.
- Установите инструменты разработчика Redux. Можете задействовать их для проверки хранилища Redux и действий в браузере.

Redux имеет предсказуемый дизайн, что упрощает разработку замечательных инструментов для отладки. Разработчики, такие как Дэн Абрамов (Dan Abramov) и др., которые работают над библиотеками Redux и React, помогли создать несколько мощных инструментов для работы с приложениями Redux. Поскольку состояние в Redux изменяется предсказуемо, можно отлаживать его по-новому: отслеживать отдельные изменения состояния приложения, проверять различия между изменениями и даже перематывать и воспроизводить состояние своего приложения с течением времени. Расширение Redux Dev Tools, позволяющее вам выполнять все это и многое другое, поставляется в комплекте как расширение браузера. Чтобы установить его в своем браузере, следуйте инструкциям, приведенным на странице github.com/zalmoxisus/redux-devtools-extension. На рис. 10.3 показан краткий обзор того, что доступно с помощью Redux Dev Tools.

После установки расширения вы увидите значок Dev Tools на панели инструментов браузера. На момент написания книги он появлялся в цвете, только если обнаруживал экземпляр приложения Redux в режиме разработки, поэтому в приложениях или на других сайтах, на которых Redux не настроен, расширение пока не работает. Но как только вы настроите приложение, вы увидите, что значок стал цветным. Нажатие на него откроет инструменты.

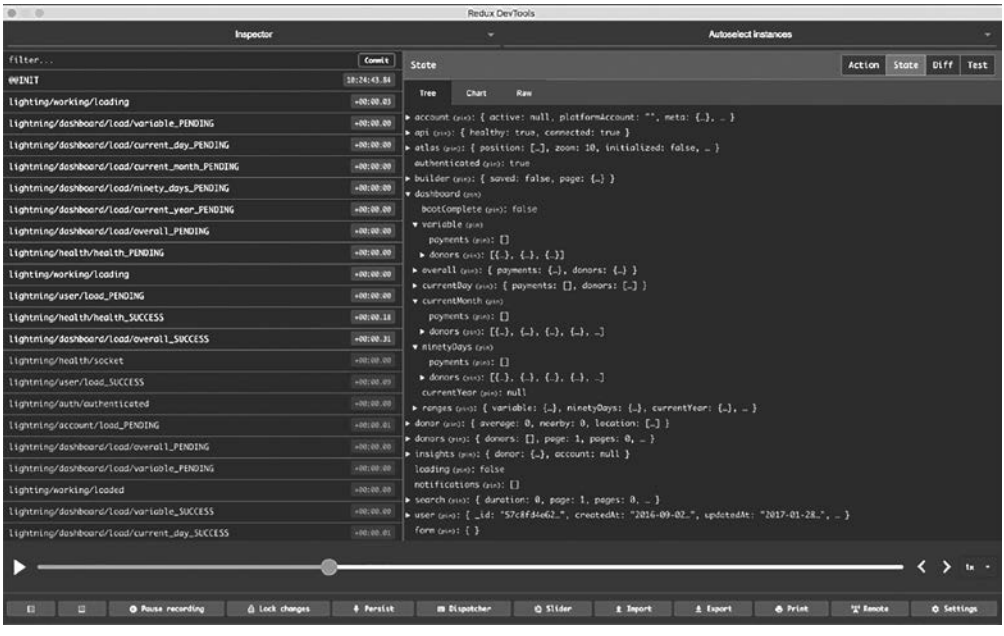


Рис. 10.3. Redux Dev Tools включает популярную библиотеку инструментов Redux Dev Tools от Дэна Абрамова в удобное расширение браузера. С его помощью вы можете перематывать и воспроизводить приложение Redux, проверять изменения одно за другим, анализировать различия между изменениями состояния, видеть состояние всего приложения в одной области, генерировать тестовый шаблон и т. д.

10.2. Действия в Redux

В Redux действия — это полезная нагрузка информацией, отправляющая данные из приложения в хранилище. Помимо действия, у хранилища нет другого способа получить данные. Действия используются во всем приложении Redux для инициирования изменений в данных, хотя сами они не несут ответственности за обновление состояния (хранилища) приложения. Редукторы больше вовлечены в эту часть архитектуры, и мы рассмотрим их после действий. Если вы привыкли обновлять состояние своего приложения, то вначале действия могут вам не понравиться. Может потребоваться время на то, чтобы к ним привыкнуть, но с ними приложения обычно более предсказуемы и легче отлаживаются. Если способ изменения данных в приложении жестко контролируется, легко предсказать, что должно или не должно было измениться в приложении. На рис. 10.4 показано, как действия вписываются в более общую картину. Мы начинаем с действий и двинемся через поток Redux, хранилище и редукторы и в итоге вернемся к React для замыкания потока данных.

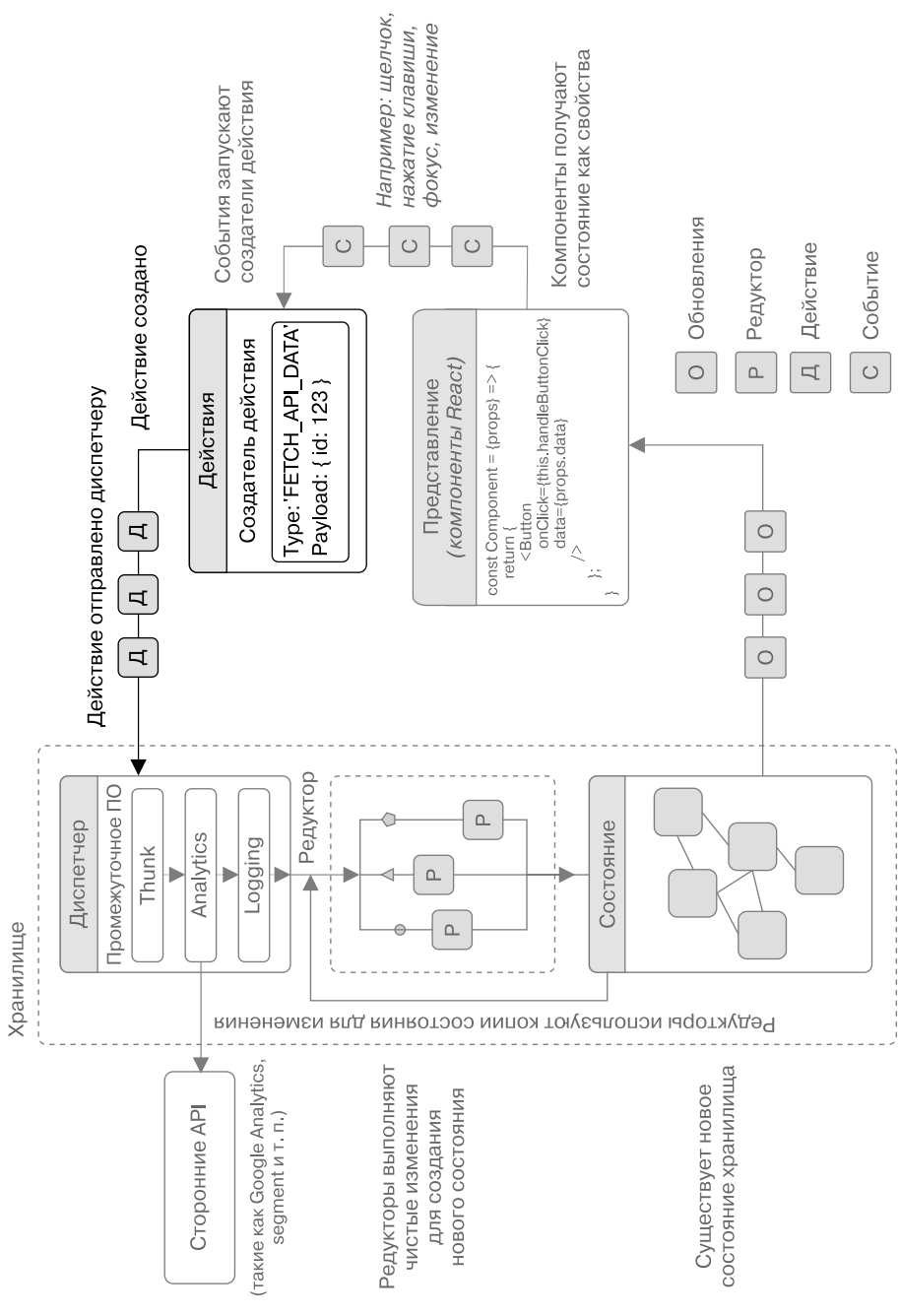


Рис. 10.4. Действия — это то, из-за чего приложение Redux знает, что нужно изменить. Они имеют тип и любую дополнительную информацию, необходимую приложению

Как выглядит действие Redux? Это простой объект JavaScript (POJO) с нужным ключом `type` и всем тем, что вы хотите в него добавить. Редукторы и другие инструменты Redux будут использовать ключ типа, чтобы связать набор изменений. Каждый уникальный тип действия должен иметь уникальный ключ типа. Типы обычно определяются как строковые константы, и вы можете давать им любые уникальные имена, которые вам нравятся, хотя придумать принцип именования и придерживаться его — это хорошая идея. В листинге 10.1 приведены несколько примеров имен типов действий.

В общем, вы должны следить за тем, чтобы в действиях содержалась только та информация, которая им совершенно необходима. Так вы избежите передачи дополнительных данных и у вас будет меньше информации, о которой нужно помнить. В листинге 10.1 показаны два простых действия: с дополнительными данными и без них. Обратите внимание на то, что при желании в действия можно ввести дополнительные ключи, но это может сбить с толку, если вы непоследовательны, и становится проблемой для команд.

Листинг 10.1. Некоторые простые действия Redux

```
{
  type: 'UPDATE_USER_PROFILE',
  payload: {
    email: 'hello@ifelse.io'
  }
}

{
  type: 'LOADING'
}

{
  type: appName/dashboard/insights/load'
}
```

Действие может содержать информацию, которая сообщит приложению о том, как оно должно измениться, например новый адрес электронной почты пользователя, сведения о диагностике ошибок и др.

Каждое действие должно иметь тип — без этого приложение не знает, какого рода изменения нужно внести в хранилище

Типы обычно представляют собой строковые константы, набранные прописными буквами, так вы отличите их от обычных значений в приложении. Здесь я использую схему пространства имен, чтобы гарантировать, что действия уникальны, но читаемы

10.2.1. Определение типов действий

Можете начать перевод своего приложения Letters Social на архитектуру Redux, определив некоторые типы действий. Скорее всего, они будут соответствовать действиям пользователя, таким как вход в систему, выход из нее, изменение значения в форме и т. д., но это не обязательно. Возможно, вы захотите создать типы действий для открытого, разрешенного или ошибочного сетевого запроса или любых других событий, которые непосредственно не относятся к пользователю.

Также стоит отметить, что в небольшом приложении не обязательно определять типы действий в файле констант — вы можете просто запомнить и передавать их, когда создаете действия, или жестко прописать в коде. Недостаток этого приема в том, что по мере развития приложения отслеживание типов действий будет доставлять неприятности и способно усложнить отладку или реорганизацию. В большинстве случаев в реальности вы определяете свои действия, то же самое сделайте и здесь.

Вы набросаете несколько типов предполагаемых действий, но позже по мере необходимости сможете свободно добавлять или удалять их. Примените подход пространства имен к именованию типов действий, но помните, что, когда создаете собственные действия, можете отслеживать их по любому шаблону, который считаете наилучшим, пока имена уникальны. Вы объедините подобные типы действий в объектах и так же легко выделите и экспортируете их как отдельные константы. Преимущество объединения заключается в том, что можно сгруппировать типы и использовать более короткие имена (`GET`, `CREATE` и т. д.), не задействуя их в самих именах переменных (`UPDATE_USER_PROFILE`, `CREATE_NEW_POST` и т. д.). В листинге 10.2 показано, как создавать начальные типы действий. Поместите их в `src/constantants/types.js`. Прямо сейчас вы разработаете все действия, которые понадобятся для этой главы, чтобы иметь возможность ссылаться на них, не возвращаясь к файлу постоянно.

Листинг 10.2. Определение типов действий (src/contstants/types.js)

```
export const app = {
  ERROR: 'letters-social/app/error',
  LOADED: 'letters-social/app/loaded',
  LOADING: 'letters-social/app/loading'
};

export const auth = {
  LOGIN_SUCCESS: 'letters-social/auth/login/success',
  LOGOUT_SUCCESS: 'letters-social/auth/logout/success'
};

export const posts = {
  CREATE: 'letters-social/post/create',
  GET: 'letters-social/post/get',
  LIKE: 'letters-social/post/like',
  NEXT: 'letters-social/post/paginate/next',
  UNLIKE: 'letters-social/post/unlike',
  UPDATE_LINKS: 'letters-social/post/paginate/update'
};

export const comments = {
  CREATE: 'letters-social/comments/create',
  GET: 'letters-social/comments/get',
  SHOW: 'letters-social/comments/show',
  TOGGLE: 'letters-social/comments/toggle'
};
```

При использовании инструментов разработчика Redux эти типы действий будут отображаться на временной шкале изменений состояния приложения, поэтому группировка имен по типу URL, как в листинге 10.2, может облегчить их чтение, когда у вас много действий и типов действий. Для их разделения можно применять символ «:» (`namespace: action_name: status`) или любое другое соглашение, наиболее подходящее в данной ситуации.

10.2.2. Создание действий в Redux

Теперь, когда вы определили некоторые типы, можно работать с действиями. Вы станете повторно использовать логику из существующих частей приложения, поэтому значительная часть кода покажется вам знакомой. Сейчас хороший момент для короткой остановки: бóльшая часть приложения Redux не должна быть полным повторением любой существующей логики приложения. Надеюсь, вы способны его почистить. Основная работа по преобразованию приложения для использования в Redux может быть простым сопоставлением различных показателей его состояния с шаблонами Redux. Во всяком случае начать нужно с действий.

Действия — это то, как вы инициируете изменения состояния в приложениях Redux: нельзя просто изменить свойство напрямую, как можно было бы в других фреймворках. Действия формируются *создателями действий* — функциями, возвращающими объект действия, и отсылаются хранилищем с помощью функции `dispatch`.

Но не будем забегать слишком далеко вперед. Сначала я расскажу о самих создателях действий. Вы начнете с простого и разработаете действия, которые будут указывать приложению, когда началась и завершилась загрузка. В это время не нужно будет передавать какую-то дополнительную информацию, но я расскажу о параметризованных создателях действий. В листинге 10.3 показано, как написать два создателя действий для загружающего и загруженного действий. Чтобы все было организованным, вы поместите создатели действий в каталог действий. То же самое касается других файлов, связанных с Redux, — редукторы и хранилище получат собственные каталоги.

Листинг 10.3. Создатели загружающих и загруженных действий (src/actions/loading.js)

```
import * as types from '../constants/types';

export function loading() {
  return {
    type: types.app.LOADING
  };
}

export function loaded() {
  return {
    type: types.app.LOADED
  };
}
```

Импортирование типов из файла констант

Возвращение объекта действия с нужным типом ключа с использованием определенного ранее загружающего типа

Экспорт создателя действия для загруженного действия

10.2.3. Создание действий для хранилища и диспетчера Redux

Создатели действий сами ничего не предпринимают, чтобы изменить состояние приложения (они просто возвращают объекты). Вам необходимо задействовать диспетчер, предоставленный Redux, чтобы деятельность создателей действий давала какой-либо эффект. Функция `dispatch` предоставляется хранилищем Redux и служит способом отправки действий в Redux для обработки. После этого вы создадите хранилище Redux, чтобы иметь возможность использовать его функцию `dispatch` в своих действиях.

Прежде чем настроить хранилище, создайте файл корневого редуктора, который позволит организовать правильное хранилище, — редуктор ничего не сделает до тех пор, пока вы не обратите внимание на редукторы и не настроите их. Вы создадите папку с именем `reducers` в `src` и файл `root.js` внутри нее. В нем применяете функцию `combReducers`, предоставляемую Redux, чтобы указать, где расположатся будущие редукторы. Эта функция объединяет несколько редукторов в один.

Не имея возможности комбинировать редукторы, вы столкнулись бы с проблемами конфликтов между несколькими из них и вынуждены были бы найти способы объединения редукторов или действий маршрута. Это одна из областей, где ощутимы преимущества Redux. Чтобы все настроить, нужно немного поработать, но как только все будет выполнено, Redux упростит масштабирование управления состоянием приложения. В листинге 10.4 показано, как создать файл корневого редуктора.

Листинг 10.4. Создание корневого хранилища (src/reducers/root.js)

```
import { combineReducers } from 'redux';
const rootReducer = combineReducers({});
export default rootReducer;
```

Импорт инструмента combReducers из Redux

Создание корневого редуктора с помощью combReducers с пустым объектом

Экспорт корневого редуктора

Теперь, когда у вас установлен редуктор для использования Redux, настройте хранилище. Сделайте папку с именем `store`, а в ней — несколько файлов: `store.js`, `stores/store.prod.js` и `stores/store.dev.js`. Они отвечают за экспорт функции, которая создает хранилище и, если вы находитесь в режиме разработки, интегрирует инструменты разработчика. В листинге 10.5 показано создание файлов, связанных с хранилищем, одного за другим. Здесь для различных сред используются разные файлы, чтобы включать различное промежуточное программное обеспечение и другие библиотеки для среды разработки и производства. Это лишь соглашение — Redux все равно, размещаете вы функции в множестве файлов или в одном.

Теперь, когда хранилище настроено и готово к работе, можете отправить в него некоторые действия и посмотреть, как они работают. Вскоре вы подключите Redux к React, но помните, что не обязаны использовать Redux с React или с какими-либо

библиотекой или фреймворком. Существуют и другие проекты с открытым исходным кодом, применяющие Redux для интеграции с такими фреймворками, как Angular, Vue и т. д.

Листинг 10.5. Создание хранилища Redux

```
// src/store/configureStore.js
import { __PRODUCTION__ } from 'environs';
import prodStore from './configureStore.prod';
import devStore from './configureStore.dev';
export default __PRODUCTION__ ? prodStore : devStore;

// src/store/configureStore.prod.js
import { createStore } from 'redux';
import rootReducer from '../reducers/root';

let store;
export default function configureStore(initialState) {
  if (store) {
    return store;
  }
  store = createStore(rootReducer, initialState);
  return store;
}

// src/store/configureStore.dev.js
import thunk from 'redux-thunk';
import { createStore, compose} from 'redux';
import rootReducer from '../reducers/root';

let store;
export default initialState => {
  if (store) {
    return store;
  }
  const createdStore = createStore(
    rootReducer,
    initialState,
    compose(window.devToolsExtension())
  );
  store = createdStore;
  return store;
};
```

Этот файл упрощает использование хранилища в приложении: не нужно определять, разрабатывать ли хранилище или создать его в рабочей среде

Передача исходного состояния в конфигурацию для применения Redux

Использование метода Redux createStore для создания хранилища

Импорт из Redux утилиты compose, которая позволит комбинировать промежуточное ПО

Убедитесь, что вы все время используете одно и то же хранилище, — это гарантирует, что вы вернете его, если другой файл обратится к уже созданному хранилищу

Если расширение инструментальных средств разработчика установлено, входим в него

У хранилища Redux есть несколько важных функций, задействуемых во время работы с Redux: `getState` и `dispatch`. `getState` захватывает моментальный снимок состояния хранилища Redux в определенный момент времени, а `dispatch` определяет, как вы будете отправлять действия в хранилище Redux. При вызове метода отправки передается действие, которое является результатом вызова создателя действия. Метод `store.dispatch()` — единственный способ вызвать изменение состояния в Redux, поэтому он работает повсюду. Затем вы попробуете использовать хранилище

для отправки нескольких действий с помощью сформированных ранее создателей действий `loading`. В листинге 10.6 показано, как отправить несколько действий с применением временного файла (`rc/store/exampleUse.js`). Он предназначен только для демонстрационных целей и не понадобится для работы основного приложения.

Листинг 10.6. Отправка действий (src/store/exampleUse.js)

```
import configureStore from './configureStore';
import { loading, loaded } from '../actions/loading';
const store = configureStore();

console.log('========== Example store ===========');
store.dispatch(loading());
store.dispatch(loaded());
store.dispatch(loading());
store.dispatch(loaded());
console.log('========== end example store ===========');
```

Импорт метода configureStore и его использование для создания хранилища

Отправка другого действия

Вызов метода dispatch хранилища и передача ему нового создателя действий; вернет объект для метода dispatch, который можно использовать

Все, что нужно сделать, чтобы отправить эти действия, — импортировать файл `exampleUse` в основной файл приложения, и он будет запущен при открытии приложения. В листинге 10.7 показаны незначительные изменения, которые необходимо внести в файл `src/index.js`. Подключив Redux к React, вы станете взаимодействовать с Redux через компоненты React и вам не нужно будет вручную отправлять действия, как делается здесь в демонстрационных целях.

Листинг 10.7. Импорт файла exampleUse (src/index.js)

```
import React from 'react';
import { render } from 'react-dom';

import { App } from './containers/App';
import { Home, SinglePost, Login, NotFound, Profile } from './containers';
import { Router, Route } from './components/router';
import { history } from './history';
import { firebase } from './backend';
import configureStore from './store/configureStore';
import initialReduxState from './constants/initialState';

import './store/exampleUse';
//...
```

Импорт файла хранилища, чтобы он запускался при открытии приложения

Загружая приложение в режиме разработки (с помощью команды `npm run dev`), вы увидите, что значок инструментальных средств Redux активен. Когда прило-

жение начнет работать, импортированный файл, который вы создали, запустится и несколько раз вызовет диспетчер хранилища, отправляя действия в хранилище. Сейчас для действий не установлено никакой обработки (с помощью редукторов), и вы ничего не перехватили для React, поэтому никаких значимых изменений не произойдет. Но если вы откроете инструменты разработчика и посмотрите историю операций, то увидите действие, отправленное и записанное для каждого из отосланных вами действий загрузки. На рис. 10.5 показаны действия, отправляемые в контексте диаграммы, и результаты, которые вы увидите в инструментах разработчика Redux.

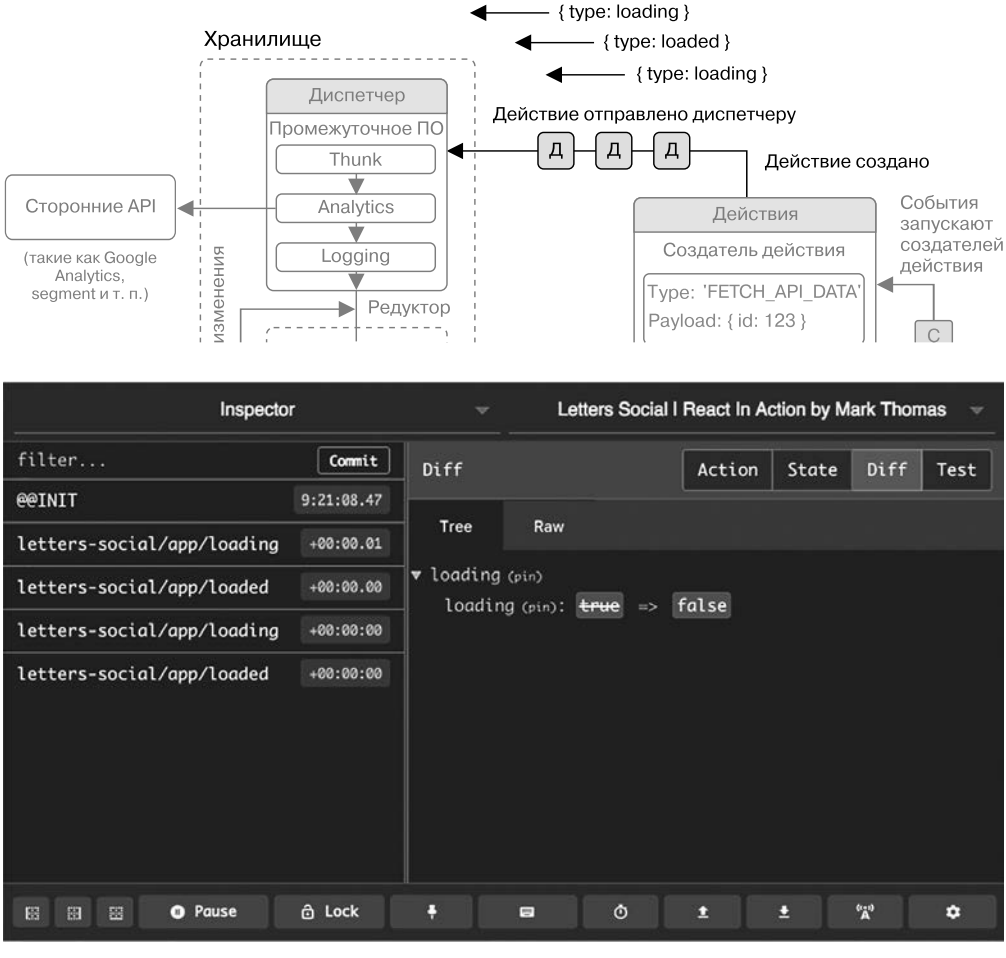


Рис. 10.5. Когда приложение будет запущено, выполненное вами хранилище примеров получит результаты работы создателей действия и отправит их в хранилище. Прямо сейчас у вас нет функционирующих редукторов, так что почти ничего не произойдет. После того как вы настроите редукторы, Redux определит, какие изменения следует выполнить в зависимости от того, какой тип действия отправлен

10.2.4. Асинхронные действия и промежуточное ПО

Вы можете отправлять действия, но сейчас они исключительно синхронны. А во многих случаях нужно вносить изменения в приложение на основе *асинхронного* действия. Это может быть сетевой запрос, считывающий значение из браузера (через локальное хранилище, хранилища cookie и т. д.), работа с WebSockets или любое другое асинхронное действие. Redux по умолчанию не поддерживает асинхронные действия, потому что ожидает, что действия будут просто объектами (а не обещаниями или чем-то еще). Но вы можете включить их, интегрировав библиотеку `redux-thunk`, которую уже установили.

`redux-thunk` — это библиотека *промежуточного* программного обеспечения Redux, то есть она работает как своего рода попутчик или проходной механизм для Redux. Вероятно, вы задействовали другие API, использующие эту концепцию, наподобие Express или Koa (серверные фреймворки для Node.js). Промежуточное ПО работает, позволяя вам вписываться в какой-то цикл или процесс таким образом, что вы создадите и примените в одном проекте несколько независимых друг от друга функций промежуточного программного обеспечения.

Согласно документации Redux, промежуточное ПО Redux является «сторонней точкой расширения между отправкой действия и моментом, когда оно достигает редуктора». Это означает, что у вас есть одна или несколько возможностей «воздействовать на» или «потому что» для действия, перед тем как его обработает редуктор. Вы будете использовать промежуточное ПО Redux для выработки решения по обработке ошибок, но прямо сейчас можете с помощью промежуточного программного обеспечения `redux-thunk` задействовать создание асинхронных действий в приложении. В листинге 10.8 показано, как интегрировать промежуточное программное обеспечение `redux-thunk` в приложение. Обратите внимание на то, что нужно добавить промежуточное программное обеспечение в свои хранилища как производства, так и разработки (`configureStore.prod.js` и `configureStore.dev.js`). Помните: вы можете выбрать любую конфигурацию хранилища производства/разработки, которая имеет наибольший смысл в вашей ситуации, — я только разбил их на две части, чтобы четко определить, какая из них для какой среды используется.

Теперь можете добавлять создатели асинхронного действия, когда у вас установлено промежуточное ПО `redux-thunk`. Почему я говорю *«создатели асинхронного действия»*, а не *«асинхронные действия»*? Потому, что, даже когда вы выполняете асинхронные операции, такие как создание сетевых запросов, создаваемые действия не являются самими асинхронными задачами. Вместо этого `redux-thunk` «обучает» хранилище Redux оценивать промисы (Promise) по мере прохождения. Суть промиса — то, как вы отправляете действия в свое хранилище. На самом деле в Redux ничего не изменилось. Действия по-прежнему синхронны, но теперь Redux знает, что нужно ждать, пока промисы будут разрешены, когда вы передадите их в функцию отправки.

В предыдущих главах вы разработали некоторую логику для извлечения сообщений из API с помощью библиотеки `isomorphic-fetch` и отображения их с помощью React. Подобные действия, которые выполняют асинхронную работу, часто требуют отправки нескольких действий (обычно это действия загрузки, успеха и отказа). Предположим, вы хотите, чтобы пользователь загружал файлы на сервер, который отправляет данные о ходе работы в течение всего времени загрузки. Одним из способов сопоставления действий с различными частями этого процесса было бы создание действия, указывающего на начало загрузки, действия, чтобы сообщить остальной части приложения о протекающей загрузке, действия для обновлений прогресса с сервера, действия для завершения загрузки и действия для обработки ошибок.

Листинг 10.8. Включение асинхронных создателей действий через redux-thunk

```
import thunk from 'redux-thunk';
import { createStore, compose, applyMiddleware } from 'redux';
import rootReducer from '../reducers/root';

let store;
export default (initialState) => {
  if (store) {
     return store;
  }
  const createdStore = createStore(rootReducer, initialState, compose(
    applyMiddleware(
      thunk,
    ),
    window.devToolsExtension()
    )
  );
  store = createdStore;
  return store;
};
```

Чтобы интегрировать промежуточное программное обеспечение в хранилище Redux, импортируется утилита applyMiddleware

Добавление и подготовка промежуточного ПО в Redux в рамках функции applyMiddleware — здесь вы вставляете промежуточное программное обеспечение redux-thunk в свое хранилище

`redux-thunk` работает, оборачивая метод отправки хранилища, и таким образом обрабатывает отправку чего-то еще, кроме простых объектов (например, промисы, API, работающего с асинхронными потоками). Промежуточное программное обеспечение будет отправлять созданные действия асинхронно (например, в начале и конце запроса), поскольку промис выполняется и позволяет соответствующим образом обрабатывать эти изменения. Как уже отмечалось, ключевым отличием здесь является то, что сами действия по-прежнему синхронны, но когда они отправляются на редукторы — асинхронны. На рис. 10.6 показано, как это работает.

Затем вы примените все знания о создателях асинхронного действия, чтобы написать несколько создателей действий, которые будут обрабатывать выборку и написание сообщений. Поскольку `redux-thunk` обертывает метод отправки хранилища, можете вернуть функцию из создателя действия, который получает метод

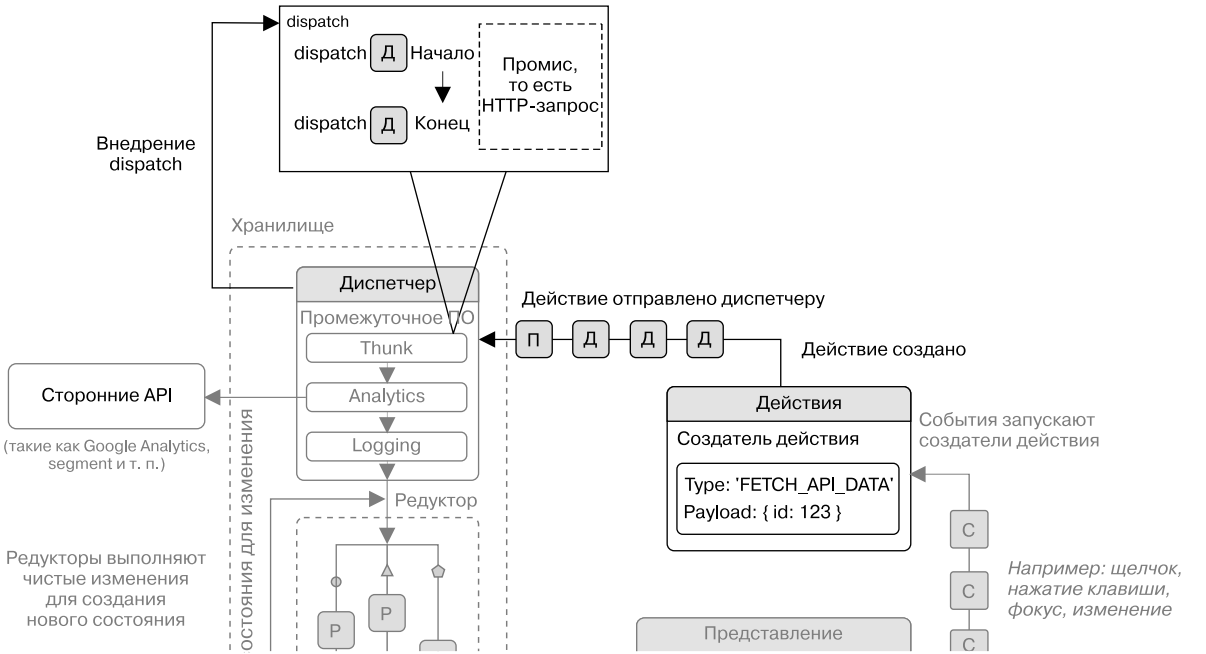


Рис. 10.6. Создатели асинхронного действия активизируются библиотекой промежуточного программного обеспечения, такой как redux-thunk, которая позволяет отправлять что-то, помимо действия типа Promise (способ выполнить асинхронную работу, которая является частью спецификации JavaScript). Она разрешит промис и позволит вам отправлять действия в разных точках в течение всего времени жизни промиса (до исполнения, по завершении, при ошибке и т. д.)

отправки как функцию, позволяя вам отправлять несколько действий в ходе выполнения Promise. В листинге 10.9 показано, как выглядит этот тип создателя действия. Вы выполните несколько создателей асинхронных действий и один — синхронных. А начнете с разработки нескольких действий, которые понадобятся для взаимодействия с пользователями, сообщениями и комментариями. Прежде всего это действие ошибки, которое будет применяться для отображения информации об ошибках пользователя, если что-то пойдет не так. В более крупном приложении вам, вероятно, потребуется разработать несколько способов обработки ошибок, но для наших целей этого должно быть достаточно. Вы можете использовать действие ошибки здесь, а также в любых границах ошибок компонента. `componentDidCatch` предоставит информацию об ошибке, которую можно отправить в хранилище.

Листинг 10.9. Создание действия ошибки (src/actions/error.js)

```
import * as types from '../constants/types';
export function createError(error, info) {
  return {
    type: types.app.ERROR,
    error,
    info
  };
}
```

Этот создатель действия параметризован, чтобы отправить информацию об ошибке в свое хранилище

Это действие имеет тип общей ошибки приложения — в крупных приложениях будет много типов ошибок

Передача совместно фактической ошибки и информации

Теперь, когда у вас есть способ обработки ошибок, можете написать несколько создателей асинхронных действий. Начните с комментариев и перейдите к сообщениям. Действия сообщений и комментариев должны выглядеть почти одинаково — с некоторыми незначительными различиями в том, как работает каждый набор действий. Можете выполнить несколько действий, связанных с комментариями: показать, скрыть и загрузить их, а также выполнить новый комментарий для данного сообщения. В листинге 10.10 показаны действия комментария, которые вы напишете.

В ходе создания этих и других действий вы продолжите использовать библиотеку `isomorphic-fetch` для выполнения сетевых запросов, но API Fetch становится более стандартным в браузерах и теперь де-факто является способом выполнения сетевых запросов. Когда возможно, задействуйте API или библиотеки платформы Web, соответствующие тем же спецификациям.

Теперь, когда разработаны действия для комментариев, можете перейти к действиям для сообщений. Они похожи на те, которые вы только что создали, и также будут задействовать некоторые действия комментариев. Возможность сочетать и сопоставлять различные действия в приложении — еще одна причина, по которой Redux применяют в качестве архитектуры приложения. Он обеспечивает структурированный, повторяемый способ создания функциональности с действиями, а затем использует эту функциональность в приложении.

Листинг 10.10. Создание действий комментариев (src/actions/comments.js)

```
import * as types from '../constants/types';
import * as API from '../shared/http';
import { createError } from './error';

export function showComments(postId) {
  return {
    type: types.comments.SHOW,
    postId
  };
}
export function toggleComments(postId) {
  return {
    type: types.comments.TOGGLE,
    postId
  };
}
export function updateAvailableComments(comments) {
  return {
    type: types.comments.GET,
    comments
  };
}
export function createComment(payload) {
  return dispatch => {
    return API.createComment(payload)
      .then(res => res.json())
      .then(comment => {
        dispatch({
          type: types.comments.CREATE,
          comment
        });
      })
      .catch(err => dispatch(createError(err)));
  };
}
export function getCommentsForPost(postId) {
  return dispatch => {
    return API.fetchCommentsForPost(postId)
      .then(res => res.json())
      .then(comments => dispatch(updateAvailableComments(comments)))
      .catch(err => dispatch(createError(err)));
  };
}
```

Импорт ваших API-помощников

Создание параметризованного создателя действия, чтобы показать определенный раздел комментариев

Вам нужна возможность переключения раздела комментариев

Реализация возможности получать комментарии — создатели асинхронного действия в файле будут использовать эту функцию

Создание комментария из данной полезной нагрузки; вместо простого объекта возвращается функция

API Fetch реализует такие методы на основе Promise, как json() и blob()

Отправка действия написания комментария с комментарием JSON, который вы получаете обратно с сервера

Если сообщается об ошибке, она отправляется в хранилище с помощью действия createError

Выбор комментариев для определенного сообщения и использование действия updateAvailableComments

Обработка ошибки, если она показана

Далее вы продолжите разрабатывать действия и добавите некоторую функциональность к сообщениям. В предыдущих главах были обеспечены функциональные возможности для получения и написания сообщений. А сейчас вы поработаете над способами отметки сообщений как понравившихся или непонравившихся. В листинге 10.11 показаны создатели действия, связанные с сообщениями в приложении. Вы начнете с четырех создателей действия, а в следующем листинге изучите еще несколько.

Листинг 10.11. Создатель асинхронных и синхронных действий (src/actions/posts.js)

```
import parseLinkHeader from 'parse-link-header';

import * as types from '../constants/types';
import * as API from '../shared/http';
import { createError } from './error';
import { getCommentsForPost } from './comments';

export function updateAvailablePosts(posts) {
  return {
    type: types.posts.GET,
    posts
  };
}
export function updatePaginationLinks(links) {
  return {
    type: types.posts.UPDATE_LINKS,
    links
  };
}
export function like(postId) {
  return (dispatch, getState) => {
    const { user } = getState();
    return API.likePost(postId, user.id)
      .then(res => res.json())
      .then(post => {
        dispatch({
          type: types.posts.LIKE,
          post
        });
      })
      .catch(err => dispatch(createError(err)));
  };
}
export function unlike(postId) {
  return (dispatch, getState) => {
    const { user } = getState();
    return API.unlikePost(postId, user.id)
      .then(res => res.json())
      .then(post => {
        dispatch({
          type: types.posts.UNLIKE,
          post
        });
      })
      .catch(err => dispatch(createError(err)));
  };
}
```

API JSON использует заголовки Link для указания параметров разбивки страницы

Как и в случае с комментариями, этот создатель действия передает новые комментарии в хранилище

Обновление ссылок на разбивку страниц в хранилище

Пометка лайком конкретного поста с использованием его идентификатора

Функция возврата обладает методами dispatch и getState, введенными в нее Redux

Отправка действия LIKE с сообщением, прикрепленным как метаданные

Пометка сообщения как непонравившегося связана с тем же потоком, но отправляет другой тип действия

Вам по-прежнему необходимо разработать несколько типов действий для сообщений. Сообщения могут нравиться или нет, но вы все еще не портировали создание сообщений, выполненное ранее. Вам также нужен способ получения нескольких сообщений и отдельных сообщений индивидуально. В листинге 10.12 показаны соответствующие создатели действий, которые вам требуется написать.

Надеюсь, к настоящему моменту вы начинаете входить во вкус работы с создателями асинхронных действий. Они довольно широко распространены во многих приложениях. Но их возможности этим не ограничиваются. Я обнаружил, что для использования большинства приложений, нуждающихся в создании асинхронных действий, достаточно лишь `redux-thunk`, однако для решения этой проблемы разработано множество других библиотек (например, посмотрите Redux Saga на странице github.com/redux-saga/redux-saga).

Листинг 10.12. Выполнение дополнительных создателей действий сообщений (src/actions/posts.js)

```
//...

export function createNewPost(post) {
  return (dispatch, getState) => {
    const { user } = getState();
    post.userId = user.id;
    return API.createPost(post)
      .then(res => res.json())
      .then(newPost => {
        dispatch({
         type: types.posts.CREATE,
         post: newPost
       });
     })
     .catch(err => dispatch(createError(err)));
  };
}
export function getPostsForPage(page = 'first') {
  return (dispatch, getState) => {
    const { pagination } = getState();
    const endpoint = pagination[page];
    return API.fetchPosts(endpoint)
      .then(res => {
        const links = parseLinkHeader(res.headers.get('Link'));
        return res.json().then(posts => {
          dispatch(updatePaginationLinks(links));
          dispatch(updateAvailablePosts(posts));
        });
      })
      .catch(err => dispatch(createError(err)));
  };
}
export function loadPost(postId) {
  return dispatch => {
    return API.fetchPost(postId)
      .then(res => res.json())
      .then(post => {
        dispatch(updateAvailablePosts([post]));
        dispatch(getCommentsForPost(postId));
      })
      .catch(err => dispatch(createError(err)));
  };
}
```

Как и ранее, используется функция getState для доступа к снимку состояния

Вставка идентификатора пользователя в новое сообщение

Отправка действия создания сообщения

Захват объекта состояния разбивки на страницы

Использование анализатора заголовка ссылки и передача заголовка Link

Отправка действия ссылки

Отправка действия обновления сообщений

Загрузка сообщения из API и получение связанных с ним комментариев

10.2.5. Redux или не Redux?

Закончив работать с создателями действий, вы подготовили начальную функциональность для написания сообщений и комментариев. Тем не менее по-прежнему не указана одна область — аутентификация пользователя. В предыдущих главах вы использовали помощников Firebase для проверки состояния аутентификации пользователя и обновления состояния локального компонента. Нужно ли сделать то же самое с аутентификацией? Тут возникает еще один хороший, но несколько спорный вопрос: что принадлежит Redux, а что — нет? Рассмотрим его, прежде чем читать дальше.

Мнения в сообществе React/Redux варьируются от «помещайте все, что вы хотите, в хранилище» до «абсолютно все должно происходить в хранилище». Также в сообществе разработчиков, которые применяли React только в контексте Redux, существует мнение, что единственно верный способ думать о React и Redux как об одном и том же. Люди часто ограничены своим опытом, но я надеюсь, что мы рассмотрим факты и компромиссы, прежде чем составить окончательное мнение.

Важно помнить, что, хотя React и Redux хорошо сочетаются, сами технологии внутренне не связаны друг с другом. Для создания React-приложений Redux не нужен. Надеюсь, здесь вы это увидели. Redux — это просто еще один инструмент, доступный для разработчиков, — это не единственный способ создать React-приложения, и, конечно же, он не ломает «нормальные» концепции React (например, состояние локального компонента). В некоторых случаях вы можете просто увеличить накладные расходы, введя состояние компонента в Redux.

Что нужно делать? Redux зарекомендовал себя как отличный способ обеспечить приложению надежную архитектуру, которая уже помогла лучше организовать код и функциональность (а мы еще не добрались до редукторов!). До сих пор вы опирались на свой опыт, поэтому у вас может возникнуть соблазн быстро согласиться с тем, что абсолютно все должно быть в хранилище Redux. Но я хочу предостеречь от всплеска эмоций и вместо этого рассмотреть компромиссы.

Мой опыт свидетельствует: есть несколько вопросов, ответы на которые могут помочь решить, что принадлежит, а что не принадлежит хранилищу Redux. Первый таков: нужно ли множеству других частей приложения знать об этом участке состояния или функциональности? Если да, вероятно, это должно быть в хранилище Redux. Если состояние полностью локализовано в компоненте, вы должны оставить его вне хранилища Redux. Одним из примеров является раскрывающееся меню, которое не нуждается ни в чьем контроле, за исключением пользователя. Если приложение должно контролировать, открыто или закрыто такое меню, и реагировать на его открытие или закрытие, эти изменения состояния должны, вероятно, проходить через хранилище. А если нет, то вполне достаточно сохранения состояния локально в компоненте.

Другой вопрос: будет ли состояние, с которым вы имеете дело, упрощено или лучше выражено в Redux? Если вы ради этого берете состояние и действия компонента и переводите их в Redux, то, вероятно, все для себя усложняете, а выгода оказывается незначительной. Но если состояние сложное или довольно детальное, так что Redux упростит его работу, можете включить его в хранилище.

Учитывая это, снова посмотрим, следует ли интегрировать логику пользователя и аутентификации в Redux. Нужно ли другим частям приложения знать о пользователе? Конечно. Способны ли вы лучше выразить логику пользователя в Redux? Без централизации в хранилище может потребоваться повторять логику на разных страницах приложения, а это далеко от идеала. Сейчас кажется, что имеет смысл интегрировать логику пользователя и аутентификации в Redux.

Посмотрим, как создать определенные действия. В листинге 10.13 показаны действия, связанные с пользователем, над которыми вы будете работать. В этих примерах используется современная функция языка JavaScript `async/await`. Если вы не знакомы с тем, как работает эта часть языка, можете прочитать документацию Mozilla Developer Network (developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/async_function) и главу об `async/await` в книге *Exploring ES2016 and ES2017* доктора Акселя Раушмайера (Axel Rauschmayer) (Leanpub, 2017; exploringjs.com/es2016-es2017/ch_asyncfunctions.html).

Листинг 10.13. Создание действий, связанных с пользователем (src/actions/auth.js)

```
import * as types from '../constants/types';
import { history } from '../history';
import { createError } from './error';
import { loading, loaded } from './loading';
import { getFirebaseUser, loginWithGithub, logUserOut, getFirebaseToken }
  from '../backend/auth';

  export function loginSuccess(user, token) {
  return {
    type: types.auth.LOGIN_SUCCESS,
    user,
    token
  };
}
export function logoutSuccess() {
  return {
    type: types.auth.LOGOUT_SUCCESS
  };
}
export function logout() {
  return dispatch => {
    return logUserOut()
      .then(() => {
        history.push('/login');
        dispatch(logoutSuccess());
        window.Raven.setUserContext();
      })
      .catch(err => dispatch(createError(err)));
  };
}
export function login() {
  return dispatch => {
    return loginWithGithub().then(async () => {
      try {
```

Импортирование модулей, необходимых для действий, связанных с аутентификацией

Подготовка создателей действий входа и выхода из системы — действие входа будет параметризовано, чтобы принять пользователя и токен

Выход пользователя из системы с помощью Firebase

Перевод пользователя на страницу входа в систему, отправка действия выхода из системы и очистка контекста пользователя (для библиотеки отслеживания ошибок)

Вход пользователя в систему с помощью Firebase

async/await использует семантику обработки ошибок try...catch

```
      dispatch(loading());
      const user = await getFirebaseUser();
      const token = await getFirebaseToken();
      const res = await API.loadUser(user.uid);
      if (res.status === 404) {
        const userPayload = {
          name: user.displayName,
          profilePicture: user.photoURL,
          id: user.uid
        };
        const newUser = await API.createUser(userPayload).then(res =>
          res.json());
        dispatch(loginSuccess(newUser, token));
        dispatch(loaded());
        history.push('/');
        return newUser;
      }
      const existingUser = await res.json();
      dispatch(loginSuccess(existingUser, token));
      dispatch(loaded());
      history.push('/');
      return existingUser;
    } catch (err) {
      createError(err);
    }
  });
};
}
```

Получение пользователя и токена из Firebase с применением await

Поиск пользователя, которого вы получили от Firebase с API. Если его не существует (404), нужно авторизовать его с помощью информации из Firebase

Создание нового пользователя

Отправка действий входа в систему с новым пользователем и возврат из функции

Если пользователь уже существует, отправка соответствующих действий входа и возврат existingUser

Отлов ошибки в процессе входа в систему и отправка ее в хранилище

В результате вы создали действия для действий, связанных с пользователем, комментариев, сообщений, загрузки и ошибок. Разработали основную часть необработанной функциональности приложения. Еще нужно научить Redux, как реагировать на изменение состояния с помощью редукторов (этим мы займемся в следующем подразделе), а затем подключить все к React. Но действия, которые вы повторно создали, представляют основные способы, которыми вы (или пользователь) можете взаимодействовать с приложением. Это еще одна сильная сторона Redux: в итоге функциональность превращается в действия и вы получаете довольно полную их коллекцию, которые можно выполнить в приложении. Это намного прозрачнее, чем базы, заполненные спагетти-кодом, где невозможно точно определить способ работы приложения, а тем более различные действия, которые можно предпринять.

10.2.6. Тестирование действий

Теперь вы напишете несколько быстрых тестов для рассмотренных ранее действий, прежде чем мы перейдем к редукторам. Я не стану описывать разработку тестов для каждого отдельного редуктора или действия, которое вы настроили, — это нецелесообразно, но хочу убедиться, что у вас есть яркие примеры, которые помогут получить представление о том, как тестировать разные части приложения Redux. Если вы хотите рассмотреть больше примеров, откройте исходный код приложения и поищите в каталоге с тестами.

Redux выполняет простое тестирование создателей действия, редукторов и других частей архитектуры Redux. Они даже лучше: их можно тестировать и поддерживать в основном независимо от интерфейсного фреймворка. Это может быть особенно важно в крупных приложениях, где тестирование является нетривиальным делом (скажем, бизнес-приложение вместо проекта выходного дня). Общим для действий является утверждение о том, что ожидаемый тип или типы действий и любая необходимая информация полезной нагрузки создаются на основе данного действия.

Большинство создателей действия можно легко протестировать, потому что они обычно возвращают объект с информацией о типе и полезной нагрузке. Иногда, однако, приходится выполнять дополнительную настройку для размещения, например, создателей асинхронных действий. Чтобы протестировать создателей асинхронных действий, используйте mock-хранилище, которое вы установили в начале главы (`redux-mock-store` — подробнее см. на странице github.com/arnaudbenard/redux-mock-store), а настраивайте его с помощью `redux-thunk`. Таким образом, можно утверждать, что создатель асинхронных действий отправляет определенные действия и проверяет, работает ли он сам, как ожидается. В листинге 10.14 показано, как протестировать действия в Redux.

Листинг 10.14. Тестирование действий в Redux (src/actions/comments.test.js)

```
jest.mock('../../src/shared/http');
import configureStore from 'redux-mock-store';
import thunk from 'redux-thunk';
import initialState from '../../src/constants/initialState';
import * as types from '../../src/constants/types';
import {
  showComments,
  toggleComments,
  updateAvailableComments,
  createComment,
  getCommentsForPost
} from '../../src/actions/comments';
import * as API from '../../src/shared/http';

const mockStore = configureStore([thunk]);
describe('login actions', () => {
  let store;
  beforeEach(() => {
    store = mockStore(initialState);
  });
  test('showComments', () => {
    const postId = 'id';
    const actual = showComments(postId);
    const expected =
     { type: types.comments.SHOW, postId };
    expect(actual).toEqual(expected);
  });
  test('toggleComments', () => {
    const postId = 'id';
```

Использование Jest для имитационного HTTP-файла, чтобы не выполнять сетевых запросов

Импорт mock-магазина и промежуточного ПО redux, чтобы создать зеркало mock-магазина

Импортирование действий, которые нужно протестировать

Импортирование API, чтобы вы могли имитировать в нем определенные функции

Создание mock-хранилища и повторная его инициализация перед каждым тестом

Утверждение о том, что создатель действия выдаст действие с правильными типом и данными

```
    const actual = toggleComments(postId);
    const expected = { type: types.comments.TOGGLE, postId };
    expect(actual).toEqual(expected);
  });
  test('updateAvailableComments', () => {
    const comments = ['comments'];
    const actual = updateAvailableComments(comments);
    const expected = { type: types.comments.GET, comments };
    expect(actual).toEqual(expected);
  });
  test('createComment', async () => {
    const mockComment = { content: 'great post!' };
    API.createComment = jest.fn(() => {
      return Promise.resolve({
        json: () => Promise.resolve([mockComment])
      });
    });
    await store.dispatch(createComment(mockComment));
    const actions = store.getActions();
    const expectedActions = [{ type: types.comments.CREATE,
      comment: [mockComment] }];
    expect(actions).toEqual(expectedActions);
  });
  test('getCommentsForPost', async () => {
    const postId = 'id';
    const comments = [{ content: 'great stuff' }];
    API.fetchCommentsForPost = jest.fn(() => {
      return Promise.resolve({
        json: () => Promise.resolve(comments)
      });
    });
    await store.dispatch(getCommentsForPost(postId));
    const actions = store.getActions();
    const expectedActions = [{ type: types.comments.GET, comments }];
    expect(actions).toEqual(expectedActions);
  });
});
```

Создание mock-комментария, чтобы передать его создателю действия

Компоновка метода createComment из модуля API с применением Jest

Отправка действия и использование await, чтобы дождаться разрешения обещания

Утверждение, что действия были выполнены такими, как ожидалось

10.2.7. Создание пользовательского промежуточного ПО Redux для отчетов о сбоях

Вы создали некоторые действия, но прежде, чем переходить к редукторам, можете добавить какие-либо из собственных промежуточных программ. *Промежуточное ПО* — это способ Redux, позволяющий подключиться к процессу передачи данных (действия, отправленные в хранилище, обработка редуктором, обновления состояния, уведомления прослушивателей). Подход Redux к промежуточному программному обеспечению такой же, как у других инструментов, таких как Express или Koa (веб-серверные фреймворки для Node.js), хотя он решает другую проблему. На рис. 10.7 показан пример потока, ориентированного на промежуточное программное обеспечение, каким он может появляться в чем-то наподобие Express или Koa.

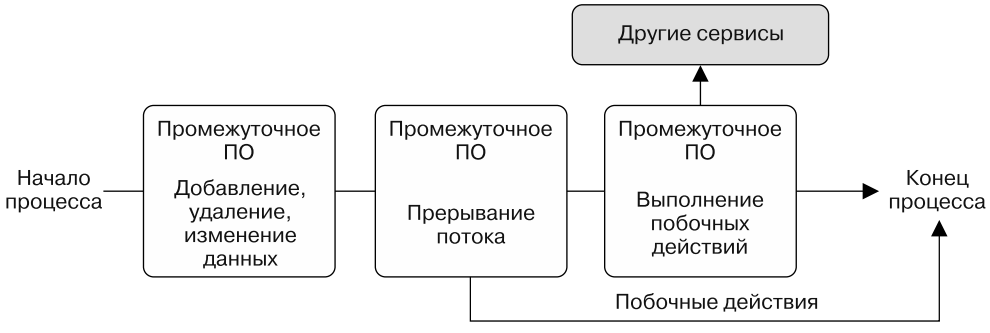


Рис. 10.7. Промежуточное программное обеспечение находится между начальной и конечной точками процесса и позволяет выполнять разные операции между ними

Иногда требуется прервать поток, отправить данные в другой API или решить иные задачи прикладного уровня. На рис. 10.7 показано несколько вариантов использования промежуточного программного обеспечения: модификация данных, прерывание потока и достижение побочных эффектов. Один из ключевых моментов заключается в том, что промежуточное программное обеспечение должно быть составным — вы должны иметь возможность менять порядок применения любого из вариантов и не беспокоиться о том, что они влияют друг на друга.

Промежуточное ПО Redux позволяет работать между точкой, в которой действие отправлено, и той, где оно достигает редуктора (см. блок «Промежуточное ПО» на рис. 10.7). Это отличное место, чтобы сосредоточиться на проблемах, общих для всех участков вашего приложения Redux, в противном случае потребовалось бы дублировать код во многих местах.

Упражнение 10.1. Определения

Сопоставьте термин с его определением:

- хранилище;
- редуктор;
- действие;
- создатель действий.

❑ Центральный объект состояния в Redux; источник истины.

❑ Объект, содержащий информацию, связанную с изменениями. Он должен иметь тип и может содержать любую дополнительную информацию, необходимую, чтобы сообщить о том, что что-то произошло.

❑ Функция, используемая Redux для вычисления изменений состояния на основе чего-то происходящего.

❑ Функция, которая применяется для создания информации о типе и полезной нагрузке, а также о том, что произошло в приложении.

Использование промежуточного программного обеспечения может быть отличным способом централизации обработки ошибок, отправки данных аналитики в сторонний API, ведения журналов и т. д. Вы внедрите простое промежуточное ПО для сообщений об ошибках, которое позволит информировать о любых необработанных исключениях систему отслеживания ошибок и управления ими. Я работаю с Sentry (sentry.io) — приложением, которое отслеживает и записывает исключения для последующего анализа, но вы можете взять то, что лучше всего подходит для вас или вашей команды (Bugsnag — еще один отличный вариант, проверьте его на bugsnag.com). В листинге 10.15 показано, как создать базовое промежуточное ПО, сообщающее об ошибках. Оно выведет ошибки, когда они встретятся Redux, и отправит их в Sentry. Как правило, разработчики получают какие-либо уведомления (сразу или на панели управления), когда в приложении появляются исключения. Sentry записывает эти ошибки и дает вам знать, когда они произошли.

Листинг 10.15. Создание простого промежуточного ПО Redux для сообщений о неполадках

```
// ... src/middleware/crash.js
import { createError } from '../actions/error';
export default store => next => action => {
  try {
    if (action.error) {
      console.error(action.error);
      console.error(action.info);
    }
    return next(action);
  } catch (err) {
    const { user } = store.getState();
    console.error(err);
    window.Raven.setUserContext(user);
    window.Raven.captureException(err);
    return store.dispatch(createError(err));
  }
};

//... src/store/configureStore.prod.js
import thunk from 'redux-thunk';
import { createStore, compose, applyMiddleware } from 'redux';

import rootReducer from '../reducers/root';
import crashReporting from '../middleware/crash';

let store;
export default function configureStore(initialState) {
  if (store) {
    return store;
  }
  store = createStore(rootReducer, initialState, compose(
    applyMiddleware(thunk, crashReporting)
  ));
  return store;
}
```

Промежуточное ПО Redux состоит из скомпонованных функций, которые будут внедряться Redux

Если ошибок нет, переход к следующему действию

Сообщение об ошибке, если таковая имеется

Получить пользователя и отослать его данные вместе с ошибкой; отправить ошибку в хранилище

Добавить промежуточное ПО, которое будет использоваться в рабочей среде

Добавить промежуточное ПО для рабочей среды

Это лишь малая часть того, что вы можете сделать с промежуточным ПО Redux. В обширной документации содержится богатая информация о Redux, рассматриваются дизайн и использование API изнутри, приведены отличные примеры. Еще более интересные примеры промежуточного ПО Redux вы найдете по адресу: redux.js.org/docs/advanced/Middleware.html#seven-examples.

10.3. Резюме

Далее представлены основные моменты, рассмотренные в этой главе.

- Redux — это библиотека и архитектура приложений, которые не обязательно должны использоваться с какими-то конкретными библиотекой или фреймворком. Он отлично работает с React и пользуется огромной популярностью в качестве инструмента управления состоянием и архитектуры приложений во многих React-приложениях.
- Redux фокусируется на предсказуемости и обеспечивает строгие способы работы с данными.
- *Хранилище* — это объект, который служит источником истины для приложения; это глобальное состояние приложения.
- Flux позволяет иметь несколько хранилищ, а Redux — только одно.
- Редукторы — это функции, используемые Redux для вычисления изменений состояния на основе данного действия.
- Redux во многом похож на Flux, но вводит идею редукторов, имеет единственное хранилище, и его создатели непосредственно не отправляют действия.
- Действия содержат информацию о том, что произошло. Они должны иметь тип, но могут содержать любую другую информацию, которая потребуется хранилищу и редукторам, чтобы определить, как должно быть обновлено состояние. В Redux существует единое дерево состояний для всего приложения; состояние живет в одной области и способно обновляться только через определенные API.
- Создатели действий — это функции, возвращающие действия, которые могут быть отправлены хранилищем. Применяя определенное промежуточное программное обеспечение (см. следующий пункт), вы можете формировать создатели асинхронных действий, предназначенные для выполнения таких действий, как вызов удаленных API.
- Redux позволяет писать промежуточное ПО — место для ввода пользовательских действий в процесс управления состоянием Redux. Промежуточное ПО выполняется до того, как будут запущены редукторы, и позволяет получить побочные эффекты или внедрить глобальные решения для приложения.

В следующей главе вы продолжите работу с Redux, узнаете о редукторах и интегрируете их в свое React-приложение.

11 Интеграция Redux и React

- Редукторы — способ Redux определить, как должно измениться состояние.
- Применение Redux с React.
- Преобразование Letters Social для использования архитектуры приложения Redux.
- Добавление в приложение функциональности лайков и комментариев.

Здесь вы продолжите работу, которой занимались в предыдущей главе, чтобы создать основные элементы архитектуры Redux. Будете работать над интеграцией React с действиями и хранилищем Redux и изучите, как работают редукторы. Redux — это вариант шаблона Flux, разработанный с учетом React, он хорошо работает с однонаправленным потоком данных и API React. Хотя это не универсальный выбор, многие крупные React-приложения рассматривают Redux как один из лучших вариантов при реализации решения для управления состоянием. Последуйте их примеру и примените его к Letters Social.

Получение исходного кода

Как и прежде, вы можете получить исходный код примеров из этой главы, перейдя в репозиторий GitHub по адресу github.com/react-in-action/letters-social. Если планируете начать работу здесь самостоятельно с нуля, возьмите исходный код примеров из глав 7 и 8 (если изучили их и сами выполнили примеры) или обратитесь к ветви, относящейся к данной главе (chapter-10-11).

Помните, что каждая ветвь содержит итоговый код главы (например, chapter-10-11 содержит код, получаемый в конце глав 10 и 11). Вы можете выполнить в оболочке командной строки одну из следующих команд по своему выбору, чтобы получить код примеров из текущей главы. Если репозитория вообще нет, выполните команду:

```
git clone git@github.com:react-in-action/letters-social.git
```

Если у вас уже клонирован репозиторий, то следующую:

```
git checkout chapter-10-11
```

Возможно, вы перешли сюда из другой главы, поэтому стоит проверить, установлены ли у вас все нужные зависимости, с помощью команды:

```
npm install
```

11.1. Редукторы определяют, как должно измениться состояние

Вы можете создавать и отправлять действия и обрабатывать ошибки, но они никак не влияют на состояние. Чтобы обрабатывать входящие действия, необходимо настроить редукторы. Помните, что действия — это просто способы сообщить, что произошло некое событие, и привести какую-то информацию о том, что произошло, не более того. Задача редукторов — указать, как изменится состояние хранилища в ответ на эти действия. На рис. 11.1 показано, как редукторы вписываются в более общую картину Redux, которую мы уже видели.

Но что такое редукторы? Если вы до сих пор наслаждались простотой Redux, то не разочаруетесь: это всего лишь простые функции, которые имеют одну цель. *Редукторы* — это чистые функции, которые принимают предыдущие состояние и действие в качестве аргументов и возвращают следующее состояние. Согласно документации Redux они называются редукторами, потому что сигнатура их метода выглядит как данные, передаваемые в `Array.prototype.reduce` (например, `[1,2,3].reduce((a, b) => a + b, 0)`).

Редукторы должны быть *чистыми* функциями, а это означает, что с учетом ввода они будут каждый раз выдавать один и тот же соответствующий вывод. Это контрастирует с действиями или промежуточным программным обеспечением, где получаются побочные эффекты и часто возникают вызовы API. Выполнение чего-либо асинхронного или нечистого (например, вызов `Date.now` или `Math.random()`) в редукторах — это антишаблон, который может ухудшить производительность или надежность приложения. Документы Redux содержат такой пункт: «Получив те же аргументы, он должен вычислить следующее состояние и вернуть его. Без сюрпризов. Никаких побочных эффектов. Без вызовов API. Никаких изменений. Просто расчет». Подробнее об этом см. redux.js.org/basics/reducers.

11.1.1. Форма состояния и начальное состояние

Редукторы начнут работать над изменением единственного магазина Redux, поэтому самое время поговорить о том, какую форму примет хранилище. Проектирование формы состояния любого приложения будет влиять на то, как работает пользовательский интерфейс приложения (и в то же время оно подвержено влиянию этой работы), но, как правило, рекомендуется хранить сырые данные по возможности отделенными от данных пользовательского интерфейса. Один из способов сделать это — хранить значения, подобные идентификаторам, отдельно от их данных и использовать идентификаторы для поиска данных.

Вы создадите исходный файл состояния, который поможет определить форму и структуру состояния. В папке констант создайте файл с именем `initialState.js`.

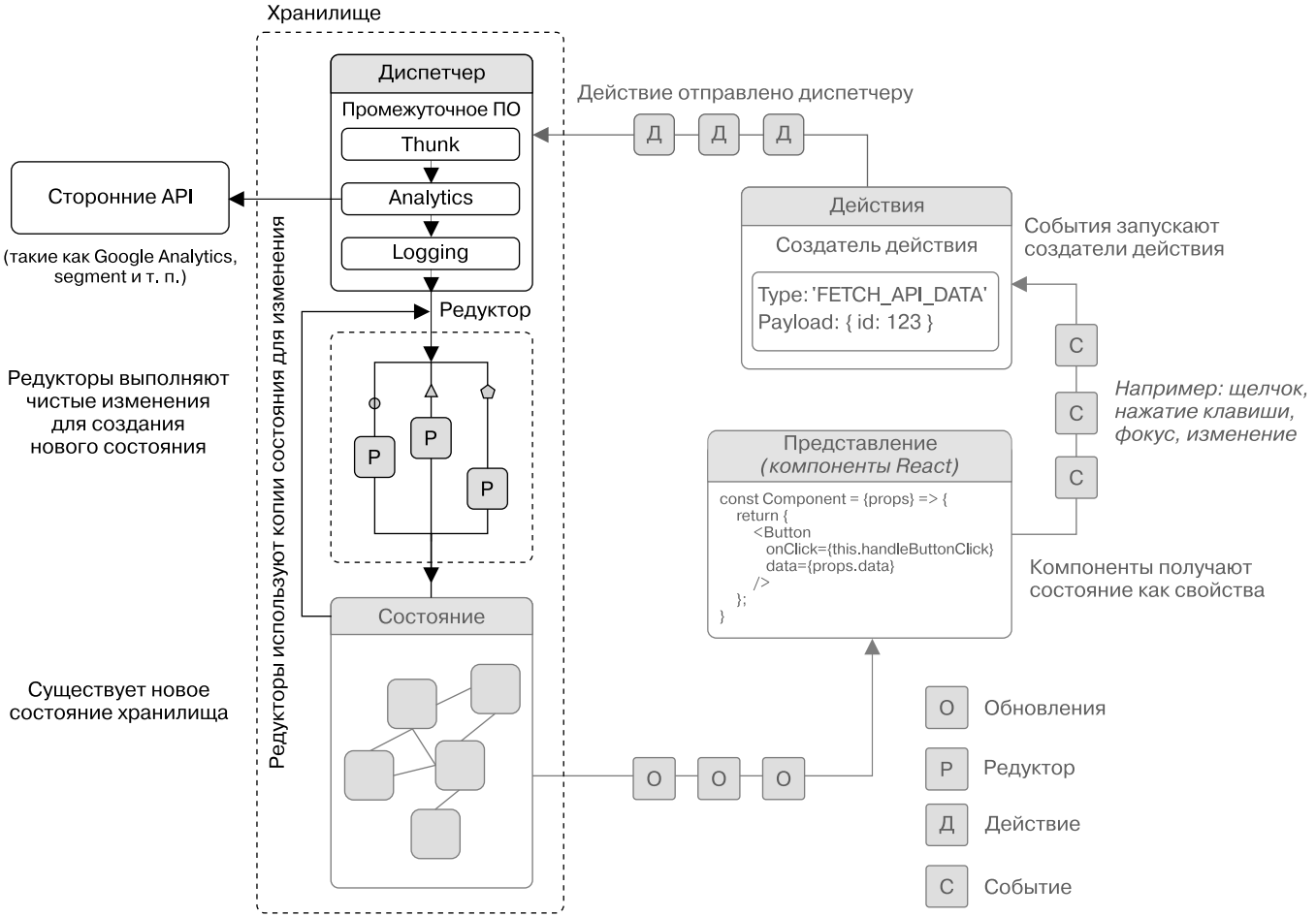


Рис. 11.1. Редукторы — это просто функции, которые помогают определить, какие изменения должны быть внесены в состояние. Рассматривайте их как своего рода шлюз для приложения, жестко контролирующий входящие изменения

Это состояние приложения Redux до того, как будут отправлены какие-либо действия или внесены изменения. Вы будете вносить в него информацию об ошибках и состояниях загрузки, а также некоторые сведения о сообщениях, комментариях и пользователе. Хранить идентификаторы для комментариев и сообщений в массивах и основную информацию для них станете в объектах, на которые легко сослаться. В листинге 11.1 показан пример настройки начального состояния.

Листинг 11.1. Начальное состояние и форма состояния (src/constants/initialState.js)

```
export default {
  error: null,
  loading: false,
  postIds: [],
  posts: {},
  commentIds: [],
  comments: {},
  pagination: {
    first: '${process.env
      .ENDPOINT}/posts?_page=1&_sort=date&_order=DESC&
        _embed=comments&_expand=user&_embed=likes',
    next: null,
    prev: null,
    last: null
  },
  user: {
    authenticated: false,
    profilePicture: null,
    id: null,
    name: null,
    token: null
  }
};
```

Объект, который Redux будет использовать для своего начального состояния

Сохранение комментариев и идентификаторов сообщений отдельно от фактических данных

Хранение ссылок на разбивку страницы (полученных через HTTP-заголовки) — это всего лишь один способ разбивки на страницы

Хранение информации о состоянии аутентификации пользователя

11.1.2. Настройка редукторов для реагирования на входящие действия

При настройке начального состояния вы должны создать несколько редукторов для обработки входящих действий, чтобы хранилище могло быть обновлено. Редукторы обычно используют инструкцию `switch`, чтобы обновить состояние в соответствии с типом входящих действий. Они возвращают новую копию состояния (не ту же самую версию с изменениями), которая затем будет применяться для обновления хранилища. Редукторы также действуют по принципу «поймать все», чтобы гарантировать, что неизвестные действия просто вернут существующее состояние. Я это уже отмечал, но важно еще раз сказать, что редукторы выполняют вычисления и должны возвращать один и тот же результат каждый раз на основе заданного ввода — никаких побочных эффектов или неясных процессов быть не должно.

Редукторы несут ответственность за расчет того, как должно измениться хранилище. В большинстве приложений у вас будет много редукторов, каждый из которых отвечает за часть хранилища. Это помогает сохранять файлы лаконично и сфокусированно. В итоге вы примените метод `combineReducers`, доступный в Redux, чтобы объединить редукторы в один. Большинство редукторов используют инструкцию `switch` со случаями для разных типов действий и команду «поймать всех» по умолчанию для всего остального, чтобы гарантировать, что неизвестные типы действий (вероятно, созданные случайно, если они есть) не повлияют на состояние.

Также редукторы делают копии состояния и не изменяют напрямую существующее состояние хранилища. Если вы посмотрите на рис. 11.1, то увидите, что редукторы используют состояние при выполнении своей работы. Такой подход аналогичен тому, как обычно работают неизменяемые структуры данных: вместо прямых изменений создаются измененные копии. В листинге 11.2 показано, как настроить редуктор загрузки. Обратите внимание на то, что в этом случае вы имеете дело с плоским срезом состояния — булевым свойством `loading`, поэтому просто возвращаете `true` или `false` для нового состояния. Вы часто будете работать с объектом состояния, у которого есть много ключей или вложенных свойств, в таком случае редуктору нужно будет сделать больше, чем просто вернуть `true` или `false`.

Листинг 11.2. Настройка редуктора загрузки (src/reducers/loading.js)

```
import initialState from '../constants/initialState';
import * as types from '../constants/types';

export function loading(state = initialState.loading, action) {
  switch (action.type) {
    case types.app.LOADING:
      return true;
    case types.app.LOADED:
      return false;
    default:
      return state;
  }}
```

Функция, принимающая два параметра — состояние и действие

Если действие имеет тип loading, возвращается true для нового значения состояния

Обработка случая loaded и возвращение соответствующего варианта false

Возвращение существующего состояния по умолчанию

Обычно используется инструкция switch для явного управления всеми типами действия и возврата состояния по умолчанию

Теперь, когда будет отправлено действие, связанное с загрузкой, хранилище Redux сможет что-то с ним сделать. Когда действие поступает и проходит через любое существующее промежуточное ПО, Redux привлекает редукторы, чтобы определить, какое новое состояние должно быть создано на основе действия. У хранилища не было способа узнать информацию об изменениях, содержащуюся в действии,

до того, как вы настроили какие-либо редукторы. Чтобы показать это, на рис. 11.2 из потока убраны редукторы; посмотрите, почему действиям не удастся достичь хранилища.

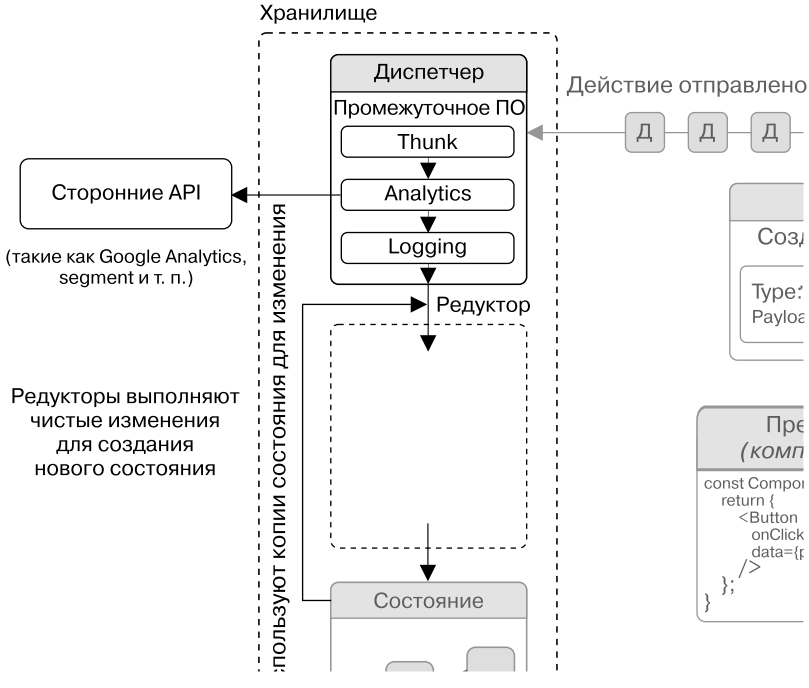
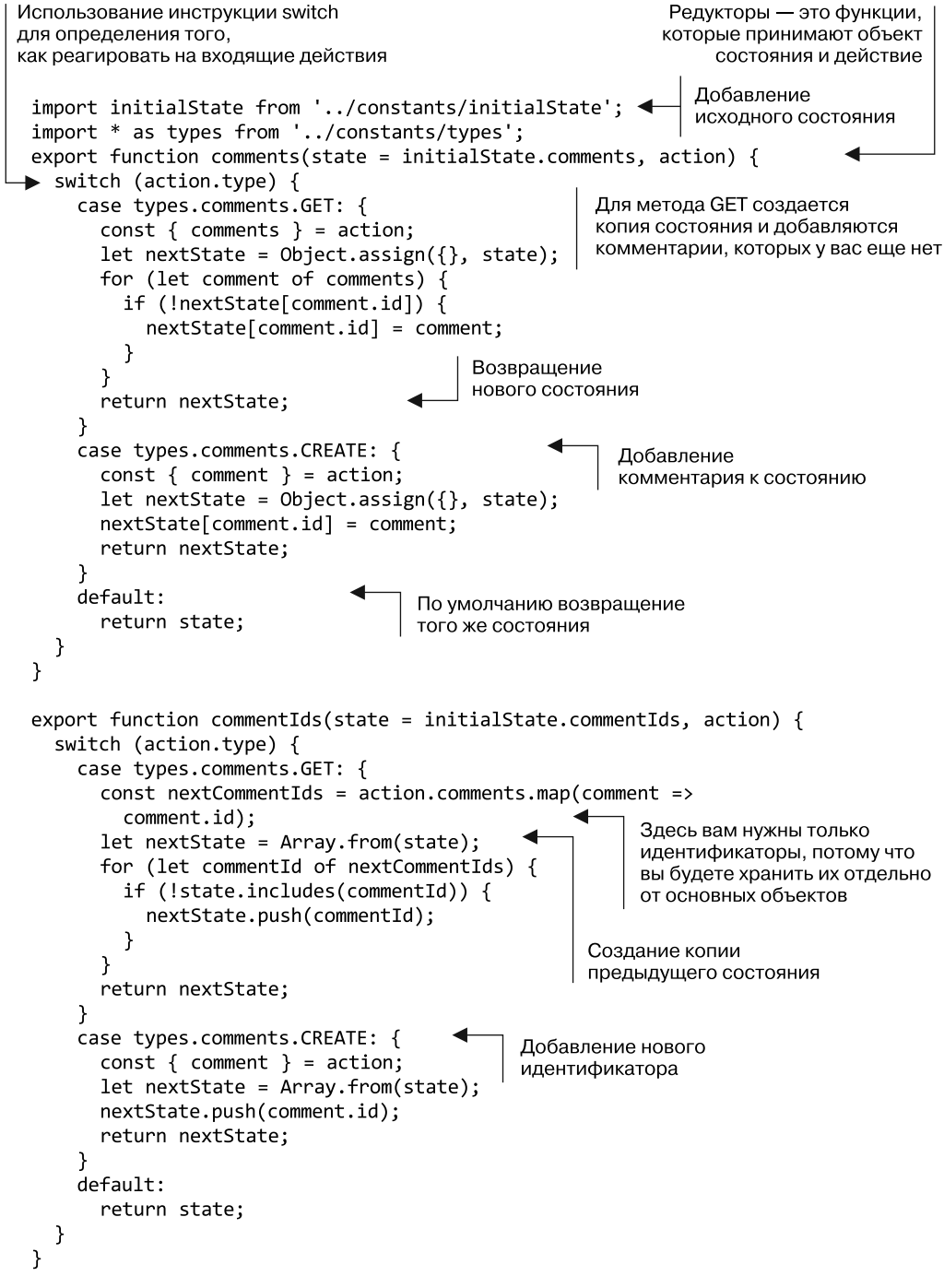


Рис. 11.2. Имея редукторы, Redux будет знать, как внести изменения в хранилище при отправке действий. В не очень сложном приложении обычно много разных редукторов, каждый из которых отвечает за свой срез состояния магазина

Затем вы создадите еще один редуктор, чтобы применить свои навыки работы с Redux. В конце концов, многие редукторы не будут просто возвращать `true` или `false`. Или по крайней мере в вычислениях будет больше того, чтобы выдать просто `true` или `false`. Другая ключевая часть приложения Letters Social показывает и создает сообщения, и нужно перенести ее в Redux. Вы должны сохранить бóльшую часть существующей логики, используемой приложением, и перевести ее в удобную для Redux форму, как было бы, если бы вы приспосабливали реальное React-приложение для применения Redux. Создайте два редуктора для обработки самих сообщений и один — для отслеживания идентификаторов сообщений. В более крупном приложении можете объединить их вместе под другим ключом, но сейчас хорошо хранить их по отдельности. Это тоже пример того, как можно настроить несколько редукторов для обработки одного действия. В листинге 11.3 показано, как написать редуктор для комментариев. Здесь вы создадите немало редукторов, и, как только это будет сделано, приложение получит не только подробное описание событий, которые могут произойти, но и способы изменения состояния.

Листинг 11.3. Создание редуктора комментариев (src/reducers/comments)




```
import initialState from '../constants/initialState';
import * as types from '../constants/types';
export function comments(state = initialState.comments, action) {
  switch (action.type) {
    case types.comments.GET: {
      const { comments } = action;
      let nextState = Object.assign({}, state);
      for (let comment of comments) {
        if (!nextState[comment.id]) {
          nextState[comment.id] = comment;
        }
      }
      return nextState;
    }
    case types.comments.CREATE: {
      const { comment } = action;
      let nextState = Object.assign({}, state);
      nextState[comment.id] = comment;
      return nextState;
    }
    default:
      return state;
  }
}

export function commentIds(state = initialState.commentIds, action) {
  switch (action.type) {
    case types.comments.GET: {
      const nextCommentIds = action.comments.map(comment =>
        comment.id);
      let nextState = Array.from(state);
      for (let commentId of nextCommentIds) {
        if (!state.includes(commentId)) {
          nextState.push(commentId);
        }
      }
      return nextState;
    }
    case types.comments.CREATE: {
      const { comment } = action;
      let nextState = Array.from(state);
      nextState.push(comment.id);
      return nextState;
    }
    default:
      return state;
  }
}
```

Теперь, когда вы отправляете действия, связанные с комментариями, состояние магазина будет соответствующим образом обновляться. Вы заметили, как можете реагировать на действия не строго одного и того же типа? Редукторы могут реагировать на действия, которые находятся в пределах их компетенции, даже если не имеют идентичного типа. Это возможно, так как, несмотря на то что срез сообщений состояния управляет сообщениями, могут существовать и другие действия, которые способны повлиять на него. Вывод: редуктор отвечает за решение вопроса о том, каким образом должен измениться конкретный показатель состояния, независимо от того, какие действие или тип действия приходят. Некоторым редукторам, возможно, потребуется знать о множестве различных типов действий, не связанных с ресурсом (сообщениями), которые они моделируют.

Теперь, создав редуктор комментариев, можете выполнить такой, который станет обрабатывать сообщения. Он будет очень похож на редуктор комментариев, потому что используется та же стратегия их хранения по отдельности, как идентификаторов и объектов. Также он должен знать, как обращаться с сообщениями с лайками и без (вы создали действия для этой функциональности в главе 10). В листинге 11.4 показано, как все это реализовать.

Листинг 11.4. Создание редукторов сообщений (src/reducers/posts.js)

```
import initialState from '../constants/initialState';
import * as types from '../constants/types';
export function posts(state = initialState.posts, action) {
  switch (action.type) {
    case types.posts.GET: {
      const { posts } = action;
      let nextState = Object.assign({}, state);
      for (let post of posts) {
        if (!nextState[post.id]) {
          nextState[post.id] = post;
        }
      }
      return nextState;
    }
    case types.posts.CREATE: {
      const { post } = action;
      let nextState = Object.assign({}, state);
      if (!nextState[post.id]) {
        nextState[post.id] = post;
      }
      return nextState;
    }
    case types.comments.SHOW: {
      let nextState = Object.assign({}, state);
      nextState[action.postId].showComments = true;
      return nextState;
    }
    case types.comments.TOGGLE: {
      let nextState = Object.assign({}, state);
```

Обработка полученных сообщений

Отображение или переключение комментариев к сообщению

```
      nextState[action.postId].showComments =
        !nextState[action.postId].showComments;
      return nextState;
    }
    case types.posts.LIKE: {
      let nextState = Object.assign({}, state);
      const oldPost = nextState[action.post.id];
      nextState[action.post.id] =
        Object.assign({}, oldPost, action.post);
      return nextState;
    }
    case types.posts.UNLIKE: {
      let nextState = Object.assign({}, state);
      const oldPost = nextState[action.post.id];
      nextState[action.post.id] = Object.assign({}, oldPost, action.post);
      return nextState;
    }
    case types.comments.CREATE: {
      const { comment } = action;
      let nextState = Object.assign({}, state);
      nextState[comment.postId].comments.push(comment);
      return state;
    }
    default:
      return state;
  }
}

export function postIds(state = initialState.postIds, action) {
  switch (action.type) {
    case types.posts.GET: {
      const nextPostIds = action.posts.map(post => post.id);
      let nextState = Array.from(state);
      for (let post of nextPostIds) {
        if (!state.includes(post)) {
          nextState.push(post);
        }
      }
      return nextState;
    }
    case types.posts.CREATE: {
      const { post } = action;
      let nextState = Array.from(state);
      if (!state.includes(post.id)) {
        nextState.push(post.id);
      }
      return nextState;
    }
    default:
      return state;
  }
}
```

Пометка сообщения «понравилось/не понравилось» включает обновление определенного сообщения в состоянии с новыми данными API

Обработка новых идентификаторов таким же образом, как и для комментариев

Я включил в эти файлы по два редуктора, потому что они очень тесно связаны друг с другом и оба действуют на одни и те же фундаментальные данные (сообщения и комментарии), но вы, вероятно, чтобы упростить работу, захотите использовать один редуктор в файле. В большинстве случаев установка вами редуктора будет отражать структуру хранилища или по крайней мере соответствовать ей. Возможно, вы заметили, что разработка формы состояния хранилища (см. начальное состояние, которое вы установили ранее в данной главе) в значительной степени влияет на то, как определяются редукторы и в меньшей степени — действия. Один вывод из этого таков: лучше потратить больше времени на разработку формы состояния, чем на придание ему лоска. Если на дизайн будет выделено слишком мало времени, вероятно, придется долго дорабатывать форму состояния, чтобы улучшить ее, тогда как тщательный дизайн и шаблоны Redux позволяют сделать добавление новой функциональности простым действием.

Миграция в Redux: нужна ли?

Я несколько раз упоминал в этой главе, что Redux может потребовать очень большой работы для первоначальной настройки (возможно, вы уже ощутили это), но в итоге она принесет свои плоды. Особенно относится это к проектам, над которыми работали я и знакомые разработчики. Один проект, в котором я участвовал, включал полную миграцию приложения из архитектуры Flux в Redux. Команда в полном составе работала около месяца, но мы смогли запустить переписывание приложения, добившись минимальной нестабильности и уменьшив количество ошибок до минимума.

А общий результат заключался в возможности более быстрого итерационного воспроизведения продукта с помощью шаблонов, которые Redux помог нам поместить в нужное место. Через несколько месяцев после миграции Redux мы закончили серию полных переработок приложения. Несмотря на то что мы переделали большую часть React-приложения, архитектура Redux позволяла внести совсем немного изменений в управление состоянием и бизнес-логику приложения. Более того, шаблоны Redux упростили внесение добавлений в состояние приложения там, где необходимо. Интеграция Redux оправдала работы по настройке и переводу на него приложения и продолжает приносить дивиденды.

Позаботившись о более сложных редукторах — редукторах для ошибок, разбивки на страницы и пользователя, — вы закончите эту часть работы с Redux. Начните с редуктора ошибок, представленного в листинге 11.5.

Затем нужно убедиться, что состояние разбивки на страницы можно обновить. Здесь разбиение на страницы относится только к сообщениям, но в более крупном приложении вам может потребоваться настроить разбиение на страницы для разных частей приложения (например, когда у вас есть сообщение со слишком большим количеством комментариев и требуется показать сразу все). Нужно всего лишь поддерживать простую разбивку на страницы для приложения, поэтому создайте редуктор разбивки на страницы (листинг 11.6).

Листинг 11.5. Создание редуктора ошибок (src/reducers/error.js)

```
import initialState from '../constants/initialState';
import * as types from '../constants/types';
export function error(state = initialState.error, action) {
  switch (action.type) {
    case types.app.ERROR:
      return action.error;
    default:
      return state;
  }
}
```

Этот срез состояния несложен, он отправляет ошибку на действие

Листинг 11.6. Создание редуктора разбивки на страницы (src/reducers/pagination.js)

Обновление URL-ссылок с новой информацией о разбивке на страницы

```
import initialState from '../constants/initialState';
import * as types from '../constants/types';
export function pagination(state = initialState.pagination, action) {
  switch (action.type) {
    case types.posts.UPDATE_LINKS:
      const nextState = Object.assign({}, state);
      for (let k in action.links) {
        if (action.links.hasOwnProperty(k)) {
          if (process.env.NODE_ENV === 'production') {
            nextState[k] =
              action.links[k].url.replace(/http:\/\//, 'https://');
          } else {
            nextState[k] = action.links[k].url;
          }
        }
      }
      return nextState;
    default:
      return state;
  }
}
```

Создание новой копии предыдущего состояния и внесение в нее URL-адреса из полезной нагрузки

Обходной маневр из-за того, как Letters Social отключает SSL при развертывании в Zeit (zeit.co/now), — игнорируйте, если вы не развертываете приложение самостоятельно

Обновление URL-адреса для каждого типа ссылки

Теперь нужно выполнить редуктор, который позволит реагировать на связанные с пользователем события, такие как авторизация и выход из системы. В этом редукторе вы также поддержите сохранение некоторых файлов cookie в браузере, чтобы использовать их в дальнейшем, когда будете выполнять рендеринг на стороне сервера в главе 12. *Cookie-файлы* — это небольшие фрагменты данных, которые сервер передает в веб-браузер пользователя. Вы работаете за компьютером каждый день, поэтому, вероятно, знакомы с cookie-файлами (вам сообщают о них на некоторых

сайтах, как того требует законодательство), но, возможно, никогда не работали с ними программным способом. Ничего страшного. Задействуйте библиотеку `js-cookie` для взаимодействия с cookie-файлами, а все, что вы будете делать с ними, — это установка и отключение одного конкретного cookie при изменении состояния аутентификации пользователя. В листинге 11.7 показано написание пользовательского редуктора для этого.

Листинг 11.7. Создание редуктора пользователя (src/reducers/user.js)

```
import Cookies from 'js-cookie';
import initialState from '../constants/initialState';
import * as types from '../constants/types';
export function user(state = initialState.user, action) {
  switch (action.type) {
    case types.auth.LOGIN_SUCCESS:
      const { user, token } = action;
      Cookies.set('letters-token', token);
      return Object.assign({}, state.user, {
        authenticated: true,
        name: user.name,
        id: user.id,
        profilePicture: user.profilePicture ||
          '/static/assets/users/4.jpeg',
        token
    });
    case types.auth.LOGOUT_SUCCESS:
      Cookies.remove('letters-token');
      return initialState.user;
    default:
      return state;
  }
}
```

Импорт библиотеки js-cookie

Извлечение пользователя и токена из действия

Сохранение токена как cookie-файла в браузере с помощью js-cookie

Возвращение копии состояния с новыми пользовательскими данными, включая токен

При выходе из системы возвращается исходное состояние пользователя и выполняется очистка cookie-файлов

11.1.3. Объединение редукторов в нашем хранилище

Наконец, следует убедиться, что редукторы интегрированы с хранилищем Redux. Несмотря на то что вы их создали, они никак не связаны. Рассмотрим корневой редуктор, разработанный в главе 10, и подумаем, как добавить к нему новые редукторы. В листинге 11.8 показано, как добавить написанные вами редукторы к корневому редуктору. Здесь важно отметить, что способ, которым `combReducers` будет создавать ключи в вашем хранилище, основывается на передаваемых редукторах. В листинге 11.8 состояние хранилища будет иметь ключи `loading` и `posts`, каждым из которых управляет соответствующий редуктор. Здесь я использую наименование свойств

ES2015, но если бы хотел, мог бы назвать итоговые ключи по-разному. Это важно отметить, чтобы вы не думали, что имена функций должны быть напрямую привязаны к ключам в хранилище.

Листинг 11.8. Добавление новых редукторов к существующему корневому редуктору (src/reducers/root.js)

```
import { combineReducers } from 'redux';

import { error } from './error';
import { loading } from './loading';
import { pagination } from './pagination';
import { posts, postIds } from './posts';
import { user } from './user';
import { comments, commentIds } from './comments';

const rootReducer = combineReducers({
  commentIds,
  comments,
  error,
  loading,
  pagination,
  postIds,
  posts,
  user
});

export default rootReducer;
```

Импортирование редукторов, чтобы их можно было добавить к корневому редуктору

combineReducers будут монтировать каждый редуктор к соответствующему ключу, но при желании вы можете изменить имена

11.1.4. Тестирование редукторов

Тестировать редукторы Redux просто из-за их чистой, независимой природы — в конце концов, это просто функции. Проверяется способность редукторов создавать определенное состояние с учетом определенного ввода. В листинге 11.9 показано, как проверить редукторы, написанные для сообщений, и отослать срезы идентификаторов состояния. Как и в других частях Redux, то, что редукторы являются также функциями, позволяет легко их изолировать и тестировать.

Листинг 11.9. Тестирование редукторов (src/reducers/posts.test.js)

```
jest.mock('js-cookie');

import Cookies from 'js-cookie';

import { user } from '../../src/reducers/user';
import initialState from '../../src/constants/initialState';
import * as types from '../../src/constants/types';

describe('user', () => {
```

mock-библиотека js-cookie

Импортирование редуктора и типов, необходимых для тестирования

```
  test('should return the initial state', () => {
    expect(user(initialState.user, {})).toEqual(initialState.user);
  });

  test('${types.auth.LOGIN_SUCCESS}', () => {
    const mockUser = {
      name: 'name',
      id: 'id',
      profilePicture: 'pic'
    };
    const mockToken = 'token';
    const expectedState = {
      name: 'name',
      id: 'id',
      profilePicture: 'pic',
      token: mockToken,
      authenticated: true
    };
    expect(
      user(initialState.user, {
        type: types.auth.LOGIN_SUCCESS,
        user: mockUser,
        token: mockToken
      })
    ).toEqual(expectedState);
    expect(Cookies).toHaveBeenCalled();
  });
  test('${types.auth.LOGOUT_SUCCESS}, browser', () => {
    expect(
      user(initialState.user, {
        type: types.auth.LOGOUT_SUCCESS
      })
    ).toEqual(initialState.user);
    expect(Cookies).toHaveBeenCalled();
  });
});
```

Утверждение, что начальное состояние будет возвращено по умолчанию

Создание имитации пользователя, токена и ожидаемого состояния для утверждения

С учетом действия авторизации проверяется, что состояние изменилось, как ожидалось

Подтверждение, что имитация cookie была вызвана

Обработка аналогичного утверждения для действия LOGOUT_SUCCESS

Мы рассмотрели бóльшую часть основ приложения Redux: хранилища, редукторы, действия и промежуточное программное обеспечение! Экосистема Redux надежна и имеет еще много областей, которые вы можете исследовать самостоятельно. Мы опустили некоторые части API и/или экосистемы Redux, такие как расширенное использование промежуточного программного обеспечения, селекторы (оптимизированные способы взаимодействия с состоянием хранилища) и многое другое. А также не стали подробно рассматривать API хранилища (например, работу с `store.subscribe()`, для того чтобы взаимодействовать с событиями обновления). Это связано с тем, что внутренняя работа с этой частью Redux будет абстрагирована с помощью библиотеки `react-redux`. (Если вам интересно углубиться в эти области и узнать больше о Redux, см. redux.js.org.) А еще в своем блоге на сайте ifelse.io/react-ecosystem я составил руководство по экосистеме React, которое включает Redux.

Упражнение 11.1. Истина или ложь

Redux — относительно небольшая библиотека для своего функционала, но она содержит несколько сильных постулатов о том, как поток данных работает в хранилище, редукторе, действиях и промежуточном программном обеспечении. Потратьте секунду, чтобы проверить, как вы разбираетесь в ней, оценив следующие утверждения.

- И | Л. Редукторы должны напрямую изменять существующее состояние.
- И | Л. Redux включает способ выполнения асинхронной работы (например, сетевых запросов).
- И | Л. Рекомендуется включить начальное состояние по умолчанию для каждого редуктора.
- И | Л. Редукторы могут быть объединены, что упрощает разделение срезов состояния.

11.2. Сведение React и Redux

Вы достигли прогресса с Redux, но пока ваши компоненты React ничего не знают о нем. Нужно каким-то образом соединить их. Теперь, когда вы настроили Redux, создав редукторы, действия и хранилище, интегрируйте свою новую архитектуру с React. Вы, наверное, заметили, что запуск Redux не требует большой работы с React. Это потому, что Redux может быть реализован без учета конкретной структуры — любой структуры вообще. Принцип работы Redux особенно хорошо подходит для React-приложений, и поэтому, по крайней мере частично, он стал одним из самых популярных вариантов архитектуры React-приложения. Но, начиная интегрировать React и Redux, помните: вы могли бы интегрировать ее с Angular, Vue, Preact или Ember.

11.2.1. Контейнеры против показательных компонентов

При интеграции Redux в React-приложение вы почти наверняка будете работать с библиотекой `react-redux`. Она является абстракцией, которая охватывает интеграцию хранилища и действий Redux в компоненты React. Я расскажу о некоторых способах использования `react-redux`, в частности о том, как привнести действия в компоненты, и рассмотрю некоторые новые типы компонентов — презентационные и контейнерные. Вам больше не нужно распространять состояние среди ваших многочисленных компонентов, потому что Redux отвечает за управление состоянием приложения посредством действий, редукторов и хранилища. Замечу еще раз: нет ничего неправильного в создании React-приложения, которое не задействует Redux, — вы получаете все прочие функции при использовании React. Предсказуемость

и добавленная структура Redux упрощают проектирование и поддержку большого и сложного React-приложения, и именно поэтому многие команды предпочитают применять его совместно с «ванильным» React.

Две новые категории компонентов — презентационные и контейнерные — являются всего лишь двумя более сфокусированными выражениями того, что компоненты уже делают. Разница между любым «старым» компонентом и компонентом представления или контейнера заключается в том, что он делает. Вместо того чтобы позволить любому компоненту обрабатывать стилизацию, данные пользовательского интерфейса и данные приложения, презентационные компоненты обрабатывают данные, относящиеся к пользовательскому интерфейсу, и сам UI, а компоненты контейнера обрабатывают данные приложения (а-ля Redux).

Важно понимать разницу между контейнерами и презентационными компонентами, но ваше приложение по-прежнему делает то же, что и раньше, улучшено лишь разделение вопросов. Вы не внесли ничего принципиально нового в приложение с Redux: компоненты React все так же будут получать свойства, поддерживать состояние, реагировать на события и визуализировать в том же жизненном цикле, что и раньше. Ключевое отличие, обеспечиваемое `react-redux`, заключается в интеграции хранилища, редукторов и действий с компонентами. И новое разделение между презентационными и контейнерными компонентами — это всего лишь образец того, как можно облегчить себе жизнь.

Рассмотрим эти два общих типа компонентов, используемых в приложении React с архитектурой Redux. Как уже отмечалось, презентационные компоненты являются компонентами «только для UI». Это означает, что они, как правило, слабо влияют на то, как данные приложения изменяются, обновляются или выпускаются.

Вот некоторые базовые свойства презентационных компонентов.

- Они определяют то, как выглядят объекты, а не управляют потоком данных.
- У них есть только собственное состояние (они являются классами React с экземпляром поддержки), если это необходимо. Большую часть времени они должны быть функциональными компонентами без состояния, которые свойства получают от Redux через привязку `react-redux`.
- Когда у них действительно есть собственное состояние, это должны быть данные, связанные с пользовательским интерфейсом, а не данные приложения, например открытое/закрытое раскрывающееся меню и его состояние.
- Они не определяют, как данные загружаются или изменяются, это должно происходить в основном в контейнерах.
- Обычно они создаются вручную, а не библиотекой `react-redux`.
- Они могут содержать информацию о стилях, задавать свойства, подобно классам CSS, определять другие связанные со стилем компоненты и любые иные данные, относящиеся к пользовательскому интерфейсу.

Если вы изучаете экосистемы React/Redux, то иногда можете увидеть ссылки на *умные* (контейнеры) и *глупые* (презентационные) компоненты. Такое обращение

к ним перестало использоваться, поскольку оказалось бесполезным и имело уничижительный смысл, но если вы заметите, что эта терминология применяется, то сможете сопоставить ее с разделением на презентации/контейнеры. С учетом этого компоненты контейнера выполняют следующее.

- Служат источником данных и, возможно, имеют состояние. Последнее обычно поступает из хранилища Redux.
- Предоставляют данные и информацию о поведении (например, действия) презентационным компонентам.
- Могут содержать другие компоненты презентации или контейнера. Для контейнера обычно являются предком со множеством презентационных дочерних компонентов.
- Чаще всего выполняются с использованием метода `react-redux` `connect` (более чем коротко) и обычно являются компонентами более высокого порядка (создающими новые компоненты из других компонентов).
- Обычно не имеют информации о стиле, которая не связана с данными приложения. Например, в срезе состояния профиля пользователя в хранилище Redux может быть указан как любимый красный цвет, но контейнер не будет использовать эти данные для какого-либо стиля — он только передаст их презентационному компоненту.

В этой главе мы будем придерживаться усредненного подхода к разбивке ваших компонентов на презентационные и соединенные, или контейнерные, компоненты. Для каждого компонента, который хотите подключить к хранилищу Redux, вы сделаете следующее.

- Измените его, экспортировав подключенный компонент в дополнение к обычному компоненту.
- Переместите любые свойства и состояние в специальные функции, которые может использовать `react-redux` (более чем коротко).
- Привнесете все необходимые действия и привяжете их к свойству `actions`, присущему компоненту.
- Где нужно, замените локальное состояние свойствами, отображаемыми на состояние хранилища Redux.

Рисунок 11.3 поможет вам лучше понять, как обычно работает подключенный компонент: имеющиеся функции Redux существенно перегруппированы вокруг компонента React, поэтому обновления из хранилища передаются компонентам.

В этой главе невозможно описать преобразования каждого компонента, упомянутого в книге, но разница между контейнерами и презентационными компонентами, а также то, как вы интегрируете Redux с React, должны обеспечить вам хорошую начальную практику и указать правильное направление дальнейших действий.

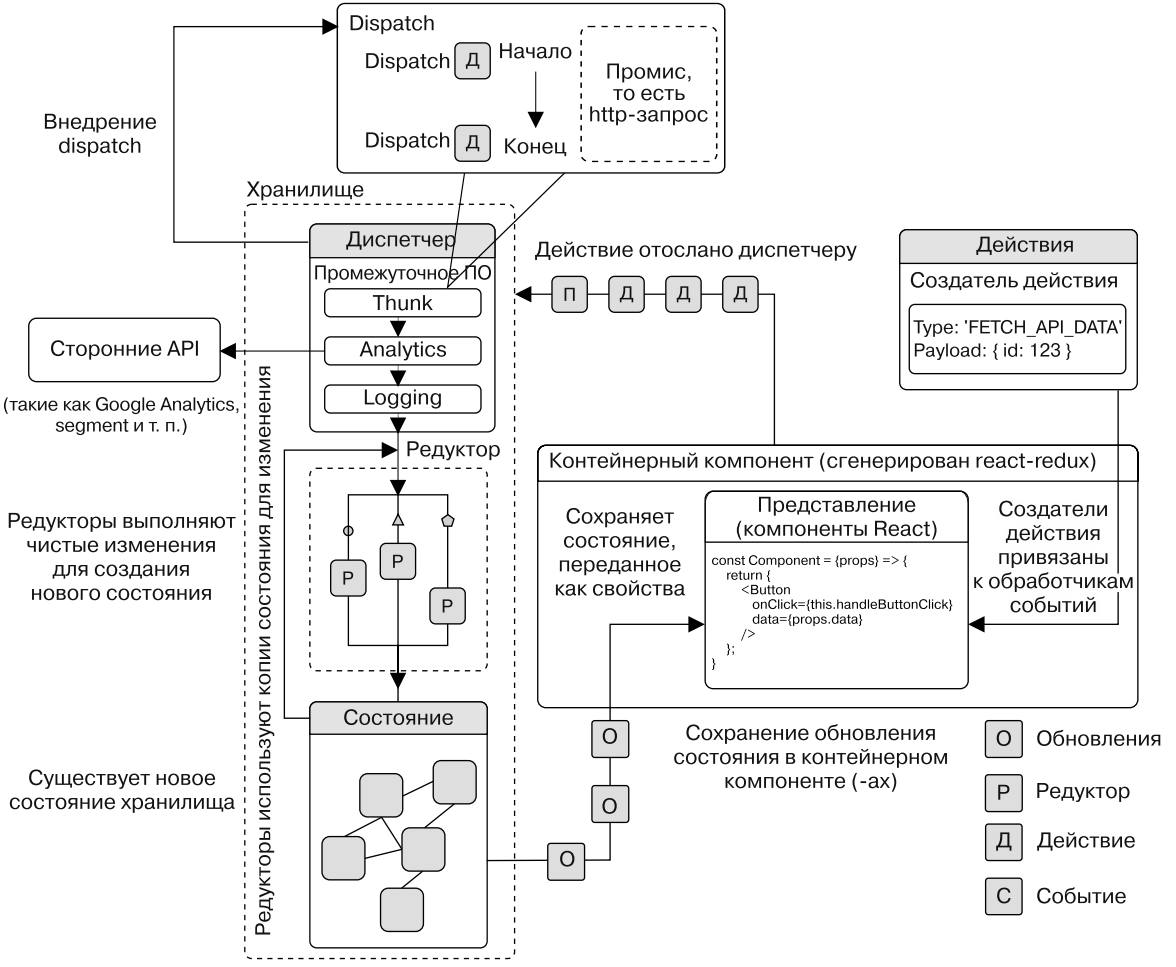


Рис. 11.3. Redux интегрирован с React. react-redux предоставляет утилиты, которые помогут вам создавать компоненты более высокого порядка, которые генерируют другие компоненты

11.2.2. Использование <Provider /> для подключения компонентов к хранилищу Redux

Первый шаг в интеграции Redux в React-приложение заключается в обертывании всего приложения компонентом Provider, предоставляемым `react-redux`. Этот компонент принимает хранилище Redux как свойство и делает его доступным для присоединенных компонентов — это другой способ описания компонентов, подключенных к Redux. Почти всегда это центральная точка интеграции между компонентами React и Redux. Хранилище должно быть доступно для контейнеров, иначе приложение не станет функционировать должным образом (или, возможно, вообще). В листинге 11.10 показано, как использовать компонент `Provider` и обновить слушатель аутентификации для обработки ваших действий Redux.

Листинг 11.10. Обертывание приложения с помощью <Provider /> react-redux

```
import React from 'react';
import { render } from 'react-dom';
import { Provider } from 'react-redux';
import Firebase from 'firebase';

import * as API from './shared/http';
import { history } from './history';

import configureStore from './store/configureStore';
import initialReduxState from './constants/initialState';

import Route from './components/router/Route';
import Router from './components/router/Router';
import App from './app';
import Home from './pages/home';
import SinglePost from './pages/post';
import Login from './pages/login';
import NotFound from './pages/404';

import { createError } from './actions/error';
import { loginSuccess } from './actions/auth';
import { loaded, loading } from './actions/loading';
import { getFirebaseUser, getFirebaseToken } from './backend/auth';

import './shared/crash';
import './shared/service-worker';
import './shared/vendor';
import './styles/styles.scss';

const store = configureStore(initialReduxState);

const renderApp = (state, callback = () => {}) => {
  render(
    <Provider store={store}>
      <Router {...state}>
        <Route path="" component={App}>
          <Route path="/" component={Home} />
```

Импортирование необходимых модулей, связанных с Redux

Создание хранилища Redux с использованием начального состояния

Оборачивание маршрутизатора в Provider из react-redux и передача в хранилище

```
          <Route path="/posts/:postId" component={SinglePost} />
          <Route path="/login" component={Login} />
          <Route path="*" component={NotFound} />
        </Route>
      </Router>
    </Provider>,
    document.getElementById('app'),
    callback
  );
};

const initialState = {
  location: window.location.pathname
};

// Render the app initially
renderApp(initialState);

history.listen(location => {
  const user = Firebase.auth().currentUser;
  const newState = Object.assign(initialState, { location: user ?
    location.pathname : '/login' });
  renderApp(newState);
});

getFirebaseUser()
  .then(async user => {
    if (!user) {
      return history.push('/login');
    }
    store.dispatch(loading());
    const token = await getFirebaseToken();
    const res = await API.loadUser(user.uid);
    if (res.status === 404) {
      const userPayload = {
        name: user.displayName,
        profilePicture: user.photoURL,
        id: user.uid
      };
      const newUser = await
        API.createUser(userPayload).then(res => res.json());
      store.dispatch(loginSuccess(newUser, token));
      store.dispatch(loaded());
      history.push('/');
      return newUser;
    }
    const existingUser = await res.json();
    store.dispatch(loginSuccess(existingUser, token));
    store.dispatch(loaded());
    history.push('/');
    return existingUser;
  })
  .catch(err => createError(err));
//...
```

Слушатель истории остается неизменным

Получение пользователя от Firebase и отправка действия loading

Создание нового пользователя, если его у вас еще нет, и отправка пользователя/токена

Загрузка существующего пользователя и отправка

Теперь, когда хранилище доступно компонентам, можете подсоединить их к своему хранилищу. Рисунок 11.3 напоминает, что `react-redux` будет вводить состояние хранилища в ваши компоненты в качестве свойств и изменять эти свойства, когда хранилище обновится. Если бы вы не использовали `react-redux`, то пришлось бы вручную подписать на обновления из хранилища все компоненты.

Чтобы это произошло, нужно задействовать утилиту `connect` из `react-redux`. Она будет генерировать контейнерный компонент, подключенный (отсюда и название) к хранилищу Redux, и при изменении последнего применять обновления. У метода `connect` всего несколько аргументов, но есть кое-что большее (подробнее об этом говорится на github.com/reactjs/react-redux). Вы будете использовать возможности как подписаться на хранилище, так и ввести функцию `dispatch` хранилища, чтобы создавать действия для своих компонентов.

Чтобы ввести состояние, вы передадите функцию (`mapStateToProps`), которая получит `state` как параметр и вернет объект, который будет включен в свойства для компонента. `react-redux` повторно вызовет эту функцию, когда компонент получит новые свойства. После того как вы используете `connect` для обертки компонента, нужно настроить способ применения свойств в компоненте (далее я описываю действия); `state` не должно использоваться, если оно не связано с данными, присущими пользовательскому интерфейсу. Помните: все это считается самой лучшей практикой, но вовсе не означает, что не существует случаев, когда возможно размывание границ между презентационными и контейнерными компонентами. Они есть, хотя и редки. Берите наилучшие решения для своей команды и конкретной ситуации.

В листинге 11.11 показано, как задействовать `connect`, настроить способ доступа к свойствам в компоненте `Home` и преобразовать его в функциональный компонент без состояния. Вы возьмете первый из двух параметров, которые в итоге передадите для подключения, — `mapStateToProps`. Эта функция получит состояние (состояние хранилища) и может иметь дополнительный аргумент `ownProps`, который будет относиться к свойствам, переданным контейнерному компоненту. Вы не станете использовать этот параметр прямо сейчас, но API предоставит его, если потребуется.

Теперь, когда вы запустите приложение (с помощью команды `npm run dev`), то не должны столкнуться с ошибками времени выполнения. Также вы не должны видеть никаких сообщений, потому что никакие действия ничего не выполняют. Но если вы откроете инструменты разработчика React, сможете увидеть `react-redux` в работе, создающим подключенный компонент. Обратите внимание на то, как `connect` создал другой компонент, обертывающий тот, который вы передали, и дал ему новый набор свойств. За кулисами он также собирается подписаться на обновления из магазина Redux и передать их в качестве новых свойств вашему контейнеру. На рис. 11.4 показано, что вы должны видеть, пошагово открыв инструменты разработчика и приложение.

Листинг 11.11. mapStateToProps (src/pages/Home.js)

```
import PropTypes from 'prop-types';
import React, { Component } from 'react';
import { connect } from 'react-redux';
import orderBy from 'lodash/orderBy';

import Ad from '../components/ad/Ad';
import CreatePost from '../components/post/Create';
import Post from '../components/post/Post';
import Welcome from '../components/welcome/Welcome';

export class Home extends Component {
  render() {
    return (
      <div className="home">
        <Welcome />
        <div>
          <CreatePost />
          {this.props.posts && (
            <div className="posts">
              {this.props.posts.map(post => (
                <Post
                  key={post.id}
                  post={post}
                />
              ))}
            </div>
          )}
          <button className="block">
            Load more posts
          </button>
        </div>
        <div>
          <Ad url="https://ifelse.io/book" imageUrl="/static/
            assets/ads/ria.png" />
          <Ad url="https://ifelse.io/book" imageUrl="/static/
            assets/ads/orly.jpg" />
        </div>
      </div>
    );
  }
}
//...
export const mapStateToProps = state => {
  const posts = orderBy(state.postIds.map(postId => state.posts[postId]),
    'date', 'desc');
  return { posts };
};
export default connect(mapStateToProps)(Home);
```

Использование функции orderBy Lodash для сортировки сообщений

Импортирование компонентов, которые показывает страница Home

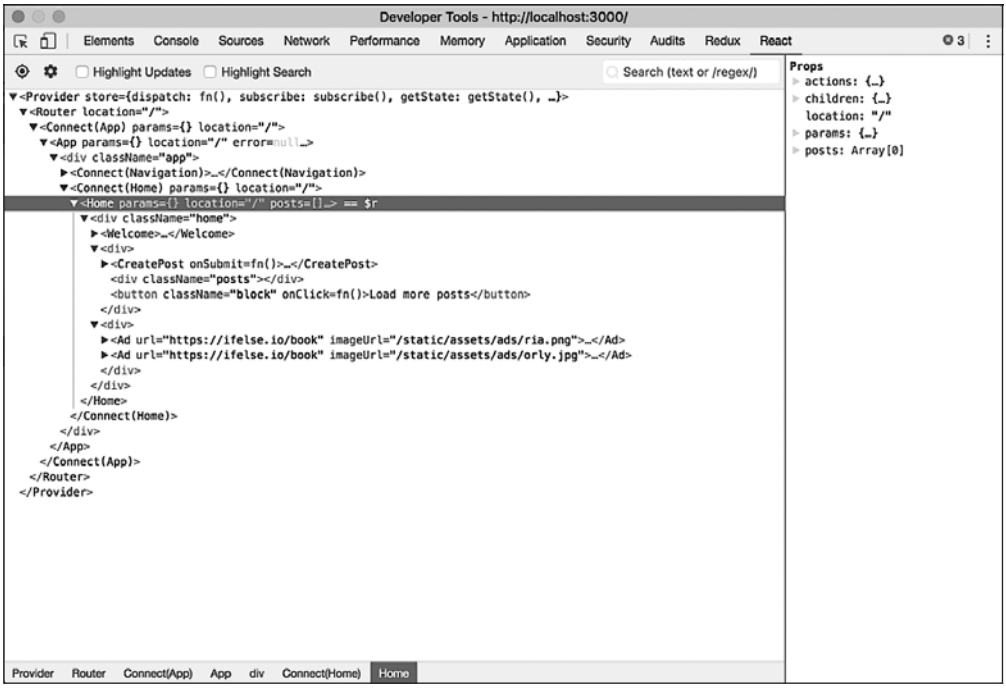
Сопоставление сообщений

Передача в сообщение и идентификатор сообщения (дальше будет обрабатываться mapStateToProps)

Сопоставление сообщений и их сортировка с помощью orderBy

Функция mapStateToProps возвращает свойства для подключенного компонента

Экспорт присоединенного компонента



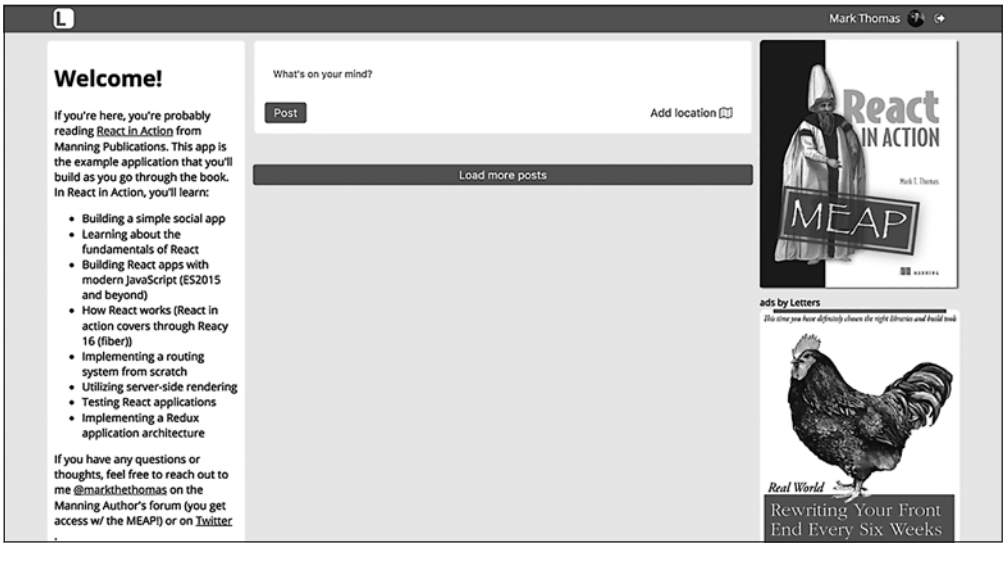


Рис. 11.4. Если вы откроете инструменты разработчика React, то сможете выбрать недавно подключенный компонент и свойства, переданные ему с помощью функции connect. Обратите внимание, как функция connect создала новый компонент, обернувший тот компонент, который вы ему передали

11.2.3. Связывание действий с обработчиками событий компонентов

Вам нужно, чтобы приложение снова реагировало на действия пользователя. Для этого примените вторую функцию — `mapDispatchToProps`. Ее аргумент `dispatch` будет методом `dispatch` хранилища, введенным в компонент. Возможно, на рис. 10.3 или в инструментах разработчика React вы заметили, что у контейнера уже есть метод `dispatch`, введенный в его свойства. Можете взять эту функцию как есть, потому что она вводится автоматически, если не задействуется функция `mapDispatchToProps`. Преимуществом функции `mapDispatchToProps` является то, что ее можно использовать для отделения логики действий компонента от самого компонента, что облегчает тестирование.

Упражнение 11.2. Назначение исходного кода

Библиотека `react-redux` предоставляет интересные абстракции, проверенные в деле многими компаниями и частными лицами, использующими Redux с React. Но вам не нужна эта библиотека, чтобы React и Redux работали вместе. В качестве упражнения потратьте некоторое время на чтение исходного кода react-redux, расположенного по адресу github.com/reactjs/react-redux/tree/master/src. Не рекомендуется создавать собственный способ соединения React и Redux, но вы должны понимать, что в этом нет ничего сверхъестественного.

Функция `mapDispatchToProps` будет вызвана `react-redux`, и полученный объект будет помещен в свойства компонентов. Вы станете использовать его, чтобы настроить свои создатели действий и сделать их доступными для компонента. А также воспользуетесь вспомогательной утилитой `bindActionCreators` от Redux. Утилита `bindActionCreators` преобразует объект, значениями которого являются создатели действий, в объект с идентичными ключами, с той разницей, что каждый создатель действия обернут в диспетчерский вызов, поэтому может быть вызван напрямую.

Рассматривая листинг 11.11, обратите внимание на то, что вместо функционального компонента без состояния использовался класс `React`. Обычно создаются функциональные компоненты без состояния, но в этом случае нужен способ начальной загрузки сообщений, поэтому вам требуются методы жизненного цикла, которые могут отправлять действия, когда компонент смонтирован. Один из способов — выгрузить события инициирования на уровень маршрутизации и согласовать данные загрузки, когда определенные маршруты вводятся или выводятся. Текущий маршрутизатор не подразумевает возможности подключения к жизненному циклу, но другие маршрутизаторы, такие как `React-router`, реализуют эту функцию. Мы рассмотрим переход к `React Router` в следующей главе, и вы воспользуетесь ее преимуществами.

Таким образом, все, что осталось, — применить функцию `mapDispatchToProps` для внедрения действий и привязки их к компонентам. Вы также можете создать объект с функциями, назначенными для любого подходящего ключа. Этот шаблон способен упростить непосредственную ссылку на действия, если в объекте `mapDispatchToProps` между функциями и вызовом отправки нет никакой дополнительной логики. В листинге 11.12 показано, как настраивать действия с помощью `mapDispatchToProps`.

Листинг 11.12. Использование mapDispatchToProps (src/container/Home.js)

```
//...
import { createError } from '../actions/error';
import { createNewPost, getPostsForPage } from '../actions/posts';
import { showComments } from '../actions/comments';
import Ad from '../components/ad/Ad';
import CreatePost from '../components/post/Create';
import Post from '../components/post/Post';
import Welcome from '../components/welcome/Welcome';
export class Home extends Component {
  componentDidMount() {
    this.props.actions.getPostsForPage();
  }
  componentDidCatch(err, info) {
    this.props.actions.createError(err, info);
  }
  render() {
    return (
      <div className="home">
        <Welcome />
        <div>
          <CreatePost onSubmit={this.props.actions.createNewPost} />
          {this.props.posts && (
            <div className="posts">
              {this.props.posts.map(post => (
                <Post
                  key={post.id}
                  post={post}
                  openCommentsDrawer=
                    {this.props.actions.showComments}
                />
              ))}
            </div>
          )}
          <button className="block"
            onClick={this.props.actions.getNextPageOfPosts}>
            Load more posts
          </button>
        </div>
        <div>
          <Ad url="https://ifelse.io/book" imageUrl="/static/
            assets/ads/ria.png" />
```

Импортирование действий, которые понадобятся для этого компонента

Загрузка сообщений при монтировании компонента

Если в компоненте возникает ошибка, для ее обработки используется componentDidCatch, ошибка отправляется в хранилище

Передача действия создания сообщения компоненту CreatePost

Передача действия showComments через свойства

Передача других сообщений для загрузки

```
          <Ad url="https://ifelse.io/book" imageUrl="/static/
            assets/ads/orly.jpg" />
        </div>
      </div>
    );
  }
}
//...
export const mapDispatchToProps = dispatch => {
  return {
    actions: bindActionCreators(
      {
        createNewPost,
        getPostsForPage,
        showComments,
        createError,
        getNextPageOfPosts: getPostsForPage.bind(this, 'next')
      },
      dispatch
    )
  };
};

export default connect(mapStateToProps, mapDispatchToProps)(Home);
```

Применение функции bindActionCreators для обертывания действий в вызов отправки

Использование функции .bind(), чтобы гарантировать,что действие getPostsForPage каждый раз вызывается с аргументом next

Теперь вы подключили свой компонент к Redux! Как я уже упоминал, книги не хватит для освещения каждого компонента вашего приложения, использующего Redux. Однако радует то, что все они выполняются по одному и тому же шаблону (создайте `mapStateToProps` и `mapDispatchToProps`, экспортируйте с помощью `connect`) и вы сможете конвертировать их для взаимодействия с Redux так же, как делали для домашней страницы. Далее представлены другие компоненты, которые вы подключили к хранилищу Redux в исходниках приложения:

- `App` — src/app.js;
- `Comments` — src/components/comment/Comments.js;
- `Error` — src/components/error/Error.js;
- `Navigation` — src/components/nav/navbar.js;
- `PostActionSection` — src/components/post/PostActionSection.js;
- `Posts` — src/components/post/Posts.js;
- `Login` — src/pages/login.js;
- `SinglePost` — src/pages/post.js.

Теперь, когда все эти компоненты интегрированы, приложение будет переведено на использование Redux! Зная, как добавить цикл Redux (создатель действия,

редуктор для обработки действий и подключение любых компонентов), как бы вы добавили новую возможность, такую как профиль пользователя? Какие еще свойства могли бы придать Letters Social? К счастью, приложение Letters Social имеет множество областей для расширения и способы, применяя которые вы можете опробовать новые функции с помощью Redux.

11.2.4. Обновление тестов

Преобразовав компонент `Home` в React, вы закончили с тестами, которые ранее написали для него. Теперь вы это исправите. К счастью, основная часть логики тестирования теперь должна располагаться в другом месте, поэтому если не все, то данные тесты должны были стать проще, чем раньше. В листинге 11.13 показан обновленный тестовый файл для компонента `Home`.

Листинг 11.13. Обновление тестов компонента Home (src/container/Home.test.js)

```
jest.mock('mapbox');
import React from 'react';
import renderer from 'react-test-renderer';
import { Provider } from 'react-redux';

import { Home, mapStateToProps, mapDispatchToProps } from
  '../../src/pages/home';
import configureStore from '../../src/store/configureStore';
import initialState from '../../src/constants/initialState';

const now = new Date().getTime();
describe('Single post page', () => {
  const state = Object.assign({}, initialState, {
    posts: {
      2: { content: 'stuff', likes: [], date: now },
      1: { content: 'stuff', likes: [], date: now }
    },
    postIds: [1, 2]
  });
  const store = configureStore(state);
  test('mapStateToProps', () => {
    expect(mapStateToProps(state)).toEqual({
      posts: [
        { content: 'stuff', likes: [], date: now },
        { content: 'stuff', likes: [], date: now }
      ]
    });
  });
  test('mapDispatchToProps', () => {
    const dispatchStub = jest.fn();
    const mappedDispatch = mapDispatchToProps(dispatchStub);
    expect(mappedDispatch.actions.createNewPost).toBeDefined();
    expect(mappedDispatch.actions.getPostsForPage).toBeDefined();
```

Имитация Mapbox, так как компонент CreateComment будет пытаться его применить, подключив тестовый визуализатор из action-test-renderer

Создание начального состояния с некоторыми сообщениями

Использование начального состояния для создания хранилища

Чтобы протестировать mapStateToProps, следует проверить, что конкретное состояние даст правильные свойства

Проверка того, что все свойства функции mapDispatchToProps правильные

```
    expect(mappedDispatch.actions.showComments).toBeDefined();
    expect(mappedDispatch.actions.createError).toBeDefined();
    expect(mappedDispatch.actions.getNextPageOfPosts).toBeDefined();
  });
  test('should render posts', function() {
    const props = {
      posts: [
       { id: 1, content: 'stuff', likes: [], date: now },
       { id: 2, content: 'stuff', likes: [], date: now }
      ],
      actions: {
        getPostsForPage: jest.fn(),
        createNewPost: jest.fn(),
        createError: jest.fn(),
        showComments: jest.fn()
      }
    };
    const component = renderer.create(
      <Provider store={store}>
        <Home {...props} />
      </Provider>
    );
    let tree = component.toJSON();
    expect(tree).toMatchSnapshot();
  });
});
```

Проверка моментального снимка, чтобы убедиться, что вывод этого компонента не изменился

Проверка моментального снимка, чтобы убедиться, что вывод этого компонента не изменился

11.3. Резюме

Вот основные сведения, которые вы почерпнули из этой главы.

- Редукторы — это функции, используемые Redux для вычисления изменений состояния на основе данного действия.
- Redux во многом похож на Flux, но вводит идею редукторов, имеет единое хранилище, и его создатели действия отправляют действия не напрямую.
- Действия содержат информацию о чем-то, что произошло. Они должны иметь тип, но могут содержать и любую другую информацию, которая потребуется вашему хранилищу и редукторам, чтобы определить, как оно должно быть обновлено. В Redux существует единое дерево состояний для всего приложения. Все состояние находится в одной области и способно обновляться только через определенные API.
- Создатели действий — это функции, возвращающие действия, которые могут быть отправлены магазином. С помощью определенного промежуточного программного обеспечения (см. следующий пункт) вы можете формировать создатели асинхронных действий, которые полезны, например, для вызова удаленных API.

- Redux позволяет писать промежуточное ПО — место для ввода пользовательского поведения в процесс управления состоянием Redux. Промежуточное ПО выполняется до того, как будут запущены редукторы, и позволяет создавать побочные эффекты или внедрять глобальные решения для своего приложения.
- `react-redux` обеспечивает привязки для компонентов React, которые позволяют подключать компоненты к хранилищу, обрабатывать передачу новых свойств и проверять наличие обновлений от Redux (при изменении хранилища).
- Контейнерные компоненты — это компоненты, которые имеют дело только с данными и не связаны с UI (думайте о них лишь как о данных приложения).
- Презентационные компоненты касаются только того, что вы видите, или данных, специфичных для пользовательского интерфейса, например, открыто ли раскрывающееся меню (думайте о них как о том, что видите).
- Redux задействует шаблон однонаправленного потока данных, в котором изменения данных вычисляются редукторами в ответ на действия, и изменения применяются к хранилищу.

В следующей главе вы изучите возможности рендеринга на стороне сервера в современных веб-приложениях и начнете использовать React на сервере.

12 React на стороне сервера и интеграция React Router

- Обработка на стороне сервера с помощью React.
- Условия необходимости рендеринга приложения на стороне сервера.
- Переход к настройке маршрутизации с помощью React Router.
- Обработка аутентифицированных маршрутов с помощью React Router.
- Получение данных во время рендеринга на стороне сервера.
- Использование Redux в процессе рендеринга на стороне сервера.

Знали ли вы, что можете применять React вне браузера? Это связано с тем, что некоторые части библиотеки `react-dom` не требуют для работы среды браузера и могут выполняться в среде run.js (или почти в любой среде выполнения JavaScript с достаточной поддержкой языка). Честно говоря, большинство JavaScript-сценариев, неспецифичных для платформы, могут работать в браузере или на сервере — это исключает функции, связанные с вводом-выводом, такие как чтение файлов, или криптографию для платформы node.js. Но, поскольку платформа node.js надежна и широко распространена, все больше и больше фреймворков создают с возможностью поддержки сервера и браузера.

Это верно и для React: она поддерживает рендеринг на стороне сервера (server-side rendering, SSR) через API React DOM. Что это значит? SSR обычно представляет собой статическую разметку HTML, которую можно отправить в браузер через HTTP или другой протокол, — это по-прежнему рендеринг, но в контексте сервера. Интеграция SSR в приложение полезна при определенных обстоятельствах и не нужна в других. В этой главе мы рассмотрим исторический контекст рендеринга на стороне сервера, определим, когда имеет смысл его реализовать, интегрируем его в свое приложение Letters Social и заменим маршрутизатор, разработанный в главах 7 и 8, чтобы лучше поддерживать SSR и позволить выполнять дальнейшие улучшения. Вы примените простую версию рендеринга на стороне сервера, используя React, чтобы ознакомиться с основными концепциями.

Получение исходного кода

Как и прежде, вы можете получить исходный код примеров из этой главы, перейдя в репозиторий GitHub по адресу github.com/react-in-action/letters-social. Если вы хотите начать работу здесь самостоятельно с нуля, возьмите исходный код примеров из глав 10 и 11 (если изучили их и сами выполнили примеры) или обратитесь к ветви, относящейся к данной главе (chapter-12).

Помните, что каждая ветвь содержит итоговый код главы (например, chapter-12 содержит код, получаемый в конце главы 12). Вы можете выполнить в оболочке командной строки одну из следующих команд по своему выбору, чтобы получить код примеров из текущей главы. Если репозитория вообще нет, выполните команду:

```
git clone git@github.com:react-in-action/letters-social.git
```

Если у вас уже клонирован репозиторий, то следующую:

```
git checkout chapter-12
```

Возможно, вы перешли сюда из другой главы, поэтому стоит проверить, установлены ли у вас все нужные зависимости, с помощью команды:

```
npm install
```

12.1. Что такое рендеринг на стороне сервера

Кратко рассмотрим исторический контекст рендеринга в веб-приложениях, прежде чем начать исследовать использование React на сервере. Если вы уже знакомы с тем, как работает SSR (возможно, раньше работали с такими фреймворками, как Ruby on Rails или Laravel, или уже понимаете механику), не стесняйтесь перейти к разделу 12.4, где начнете внедрять SSR в свое приложение.

В прошлом приложения с рендерингом на стороне сервера были широко распространены (и по сей день это верно для многих из них). Обычно они создавали код HTML, чередующийся с пользовательскими или другими данными, и отправляли его в браузер по протоколу HTTP. Технологии развиваются, но поначалу даже серверный компонент был примитивным. Были разработаны простые сценарии на стороне сервера, которые вручную соединяли части строк HTML, а затем отправляли их в качестве ответа. Это работало, но все становилось сложнее, чем должно было быть, поскольку написание конкатенированных представлений вручную было трудоемким и трудноизменяемым. Со временем фреймворки и даже языки были доработаны или созданы для того, чтобы разработчикам было проще писать пользовательские интерфейсы, которые в основном рендерились на сервере.

На рис. 12.1 дан общий обзор этого процесса. Основная идея заключается в том, что серверы реагируют на запросы браузера с динамически созданным HTML, который, например, содержит информацию, специфичную для запрашивающего пользователя. На примере шаблона ERB показано, как может действовать разработчик, выполняя разметку HTML. Возможно, вы знакомы с языком шаблонов Pug (Jade при создании), если раньше работали в сообществе node.js.

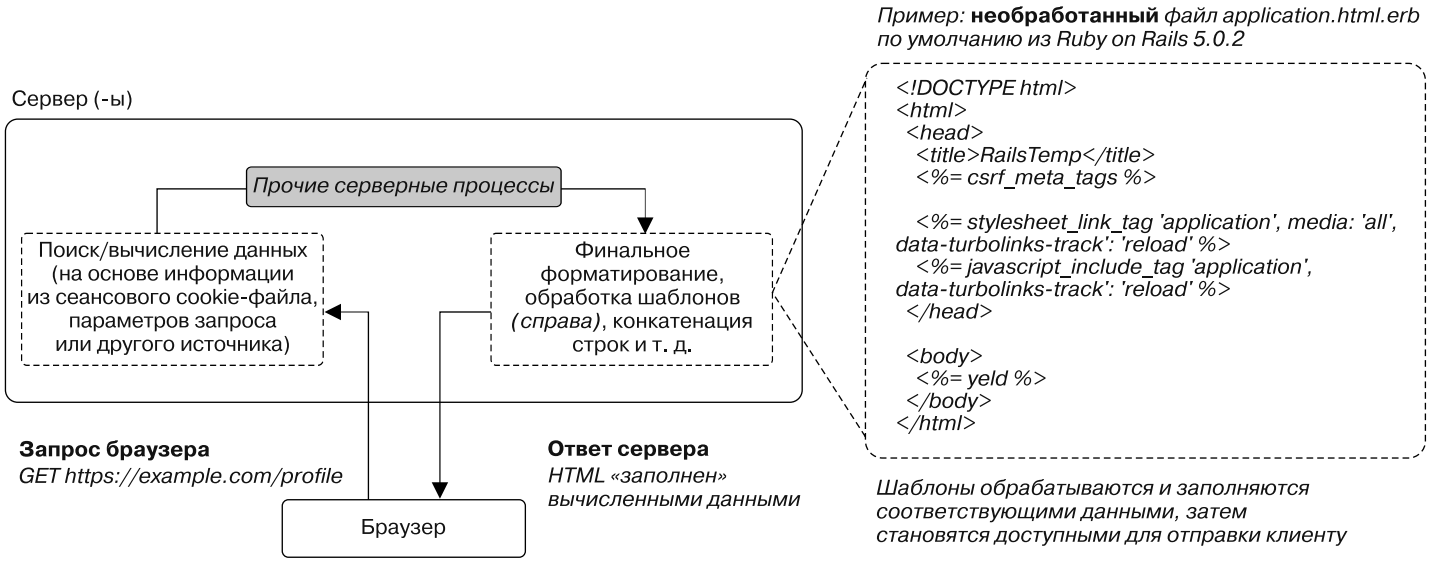


Рис. 12.1. Упрощенный обзор рендеринга на стороне сервера

Такие фреймворки, как Ruby on Rails, WordPress (основанный на PHP фреймворк управления контентом) и другие, полностью сформированные и развивающиеся, призваны удовлетворить потребность в таком создании приложений. Подход, сфокусированный на сервере, хорошо работал и по-прежнему работает. Но, поскольку клиентский JavaScript стал надежнее, а браузеры — мощнее, разработчики начали использовать JavaScript для более глобальных целей, чем просто добавление базовой интерактивности в свои приложения. С его помощью они разрабатывали и обновляли интерфейсы с динамическими данными. Это означало, что сервер меньше задействовался для шаблонов и больше — как источник данных. Сегодня вы обнаружите, что многие приложения (например, ваши) применяют надежное клиентское приложение для управления пользовательским интерфейсом и удаленным (обычно REST) API для предоставления динамических данных. Эту парадигму вы использовали в книге до сих пор. Но данная глава начинает немного ее менять, потому что вы смешиваете обработанные сервером и обработанные клиентом шаблоны. Далее будет приведен более конкретный пример того, что происходит при рендеринге на стороне сервера. На рис. 12.2 показан пример этой установки в сравнении с проиллюстрированной на рис. 12.1.

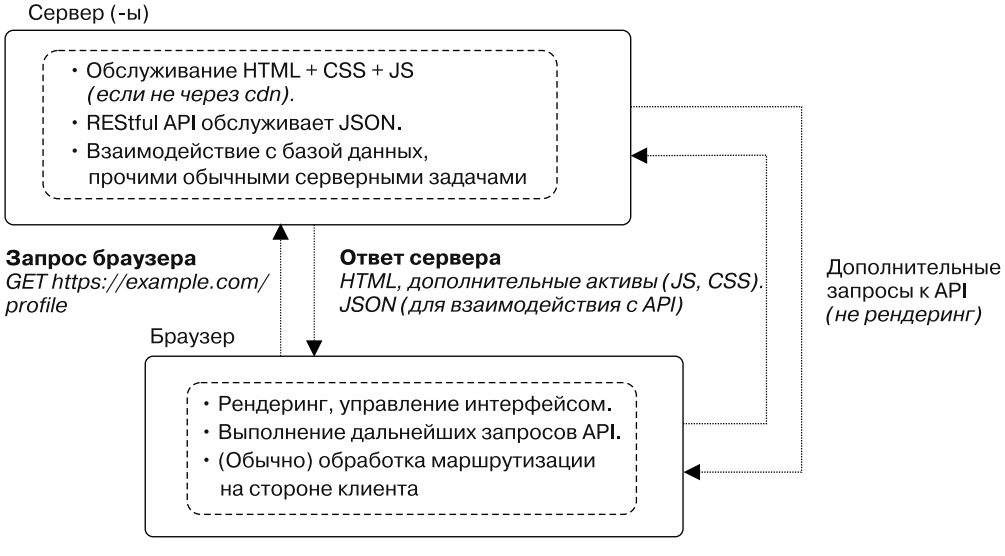


Рис. 12.2. Поскольку браузеры и JavaScript развивались (иногда медленно), у клиентского JavaScript росло количество обязанностей. Как на рис. 12.1, так и здесь выполняются одни и те же основные задачи (выборка или вычисление данных, их показ пользователю), но у клиента и сервера обязанности разные

Углубление в рендеринг на стороне сервера. Прежде чем вы начнете внедрять SSR, поговорим об особенностях его использования вне React, так что, начиная встраивать его в приложение, вы будете лучше понимать стоящую перед вами задачу. Давайте рассмотрим пример SSR, который применяет ERB (Embedded Ruby). Мы видели ERB на рис. 12.1. ERB — это одна из возможностей языка программирования Ruby,

которая может использоваться для создания шаблонов HTML (или других типов текста, например XML для генерации RSS-каналов). Если вам интересно, узнайте больше о ERB и Ruby on Rails по адресу guides.rubyonrails.org/layouts_and_rendering.html.

Многие приложения Ruby on Rails будут включать представления, созданные с помощью шаблонов ERB. Фреймворк считывает файлы шаблонов `.erb`, выполненные разработчиками, и заполняет их, используя данные с сервера или из другого места. Заполненный данными текст будет отправлен в браузер пользователя. Возможность применять шаблоны HTML-представления аналогична JSX, хотя и с другими синтаксисом и семантикой. React создает пользовательский интерфейс и управляет им, тогда как шаблонные подходы, такие как ERB, охватывают только половину процесса создания. В листинге 12.1 показан простой пример файла ERB для демонстрации типа шаблонов, который часто используется в приложениях, визуализированных сервером. Помимо различий в синтаксисе, не должно быть слишком больших различий с тем, к чему вы привыкли, работая с другими языками шаблонов, такими как Handlebars, Jade, EJS или даже React. Многие из этих языков шаблонов позволяют задействовать многие базовые конструкции, доступные на языках программирования, такие как цикл, доступ к переменной и т. д. React JSX ничем от них не отличается.

Листинг 12.1. Шаблоны ERB

```
<h1>Listing Books</h1>
<table>
  <tr>
    <th>Title</th>
    <th>Summary</th>
    <th></th>
    <th></th>
    <th></th>
  </tr>
<% @books.each do |book| %> #A
  <tr>
    <td><%= book.title %></td>
    <td><%= book.content %></td>
    <td><%= link_to "Show", book %></td>
    <td><%= link_to "Edit", edit_book_path(book) %></td>
    <td><%= link_to "Remove", book, method: :delete, data: { confirm: "Are
    you sure?" } %></td>
  </tr>
<% end %>
</table>
<br>
<%= link_to "New book", new_book_path %>
```

Возможно, было бы полезно быстро взглянуть на то, что отправляется в браузер в процессе рендеринга на сервере, чтобы получить представление о механизме, который вы хотите построить. Обработав шаблон, как показано в листинге 12.1, сервер отправляет текстовый ответ браузеру. Результат будет выглядеть примерно так, как в листинге 12.2, где показано текстовое представление ответа HTTP (версия 1/1.1). Это похоже на то, что вы будете отправлять в браузер в ходе визуализации на сервере приложения Letters Social.

Я использовал обычный консольный инструмент командной строки cURL для получения веб-страницы по адресу http://example.com, чтобы вы могли увидеть необработанный HTTP-запрос. Вероятно, у вас на компьютере уже установлен cURL, но если это не так, перейдите на страницу github.com/curl/curl и следуйте приведенным на ней инструкциям. В листинге 12.2 показан пример необработанного HTTP-ответа на запуск `curl -v https://example.com`. Я пропустил кое-что для краткости и оставил символы > и < из cURL, чтобы обозначить исходящие (>) и входящие (<) сообщения. Если вы не хотите применять cURL, всегда можете перейти по адресу http://example.com в браузере и открыть инструменты разработчика. В Chrome, Firefox и Edge есть сетевые разделы, которые позволяют также проверять HTTP-запросы.

Листинг 12.2. Пример HTTP-запроса

```
> GET / HTTP/1.1
> Host: example.com
> User-Agent: curl/7.51.0
> Accept: */*

< HTTP/1.1 200 OK
< Cache-Control: max-age=604800
< Content-Type: text/html
< Date: Mon, 01 May 2017 16:34:13 GMT
< Etag: "359670651+gzip+ident"
< Expires: Mon, 08 May 2017 16:34:13 GMT
< Last-Modified: Fri, 09 Aug 2013 23:54:35 GMT
< Server: ECS (rhv/81A7)
< Vary: Accept-Encoding
< X-Cache: HIT
< Content-Length: 1270
<
<!doctype html>
<html>
<head>
  <title>Пример домена</title>

  <meta charset="utf-8" />
  <meta http-equiv="Content-type" content="text/html; charset=utf-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1" />
</head>

<body>
<div>
  <h1>Пример домена</h1>
  <p>This domain is established to be used for illustrative examples in
  documents. You may use this
  domain in examples without prior coordination or asking for
  permission.</p>
  <p><a href="http://www.iana.org/domains/example">More
  information...</a></p>
</div>
</body>
</html>
```

Запрос, который вы отправили на сервер с помощью cURL

В заголовках ответов содержатся сведения о статусе ответа и другая полезная информация (Cache-Control, Expires и т. д.)

Тело ответа — то, для генерации чего вы будете использовать React

Вы хотите, чтобы к концу этой главы серверная часть приложения смогла выдать тот же результат, что и в листинге 12.2 (конечно, для вашего приложения). Надеюсь, к настоящему времени общая идея рендеринга на сервере кажется понятной. В следующих двух разделах мы исследуем, когда нужно встраивать такую функциональность в приложение, а когда это не имеет смысла.

12.2. Зачем рендерить на сервере

Почему вы хотите использовать SSR? На это у вас могут быть весьма веские причины. Например, есть неопровержимые доказательства того, что приложение, визуализированное на сервере, лучше работает, когда дело доходит до индексации и сканирования поисковыми системами. Хотя кажется, что крупные поисковые системы, такие как Google, могут выполнять или по крайней мере эмулировать JavaScript и DOM на сервере, также кажется, что сайты, которые отображают динамическое содержимое, не требуя DOM, имеют тенденцию работать лучше. Трудно установить точно, как влияют SSR и отличные от него приложения на поисковую оптимизацию (SEO), поскольку алгоритмы оценки сайта Google и других компаний очень похожи, но люди и команды, занятые в отрасли, сообщают по меньшей мере о некоторых фактах, говорящих о том, что воздействие может быть положительным. Если вы владеете очень популярным приложением, которое сильно зависит от места в результатах поисковой системы, подумайте о том, что SSR повысит удобство поиска вдобавок к другим SEO.

В этой книге вы разрабатываете приложение, которое требует интерактивности и позволяет пользователям динамически создавать содержимое, но такие требования предъявляются не к каждому приложению. Если вам нужны только статические функции React, для создания статического генератора страниц используйте возможности статического рендеринга React-DOM или библиотеку шаблонов.

Еще одна причина того, что вы, вероятно, захотите выполнить рендеринг на сервере, — это удобство пользователей. Если содержимое приложения должно быть показано пользователям максимально быстро, визуализация на сервере может позволить вам сделать это быстрее, чем продлилось бы ожидание на стороне клиента. Так может произойти, если приложение очень зависит от показа рекламы или другого статического платного контента и размер полезной нагрузки невелик. В тех случаях, когда требуется быстро показывать содержимое без взаимодействия, вы склонны больше беспокоиться о *первом отображении* — о том, когда пользователь впервые видит что-то в своем браузере.

Первое *отображение* — это один из многих показателей, которые можно использовать для определения того, насколько хорошо приложение рендерится браузером. Еще одним является *индекс скорости восприятия* (обычно просто *индекс скорости*, или *SpeedIndex*). Он рассчитывается путем записи того, какая часть страницы визуализирована с течением времени. Браузеры будут записывать видео страницы по мере загрузки и определять с заданным интервалом, какой ее процент загружен. Этот показатель может быть полезен для понимания на обобщенном уровне того,

как быстро данная страница загружается для пользователя. SSR способен помочь получить более высокий индекс скорости, позволяя браузеру визуализировать большую часть сайта раньше в ходе загрузки. Подробнее об индексе скорости читайте на странице sites.google.com/a/webpagetest.org/docs/using-webpagetest/metrics/speed-index.

Большинство приложений выиграют от более высокого индекса скорости и быстрого появления первого отображения. Но в каких-то случаях не нужно показывать что-то пользователям как можно быстрее, потому что вас больше интересует, как скоро они могут начать использовать приложение. Время до того, как пользователь сможет взаимодействовать с приложением или страницей, называемое *временем до интерактивности* (TTI), может оказаться чрезвычайно важным, если приложение является очень интерактивным и многофункциональным, таким как Basecamp или Asana. Для таких приложений SSR не обязательно имеет смысл, потому что они не являются публичными и полагаются скорее на интерактивность, чем на быстрый показ чего-либо пользователям.

Давайте рассмотрим несколько приложений и поговорим о том, как TTI может гипотетически повлиять на них.

- *Базовый лагерь (приложение для управления проектами)*. Пользователи хотят иметь возможность искать решения проблем, обновлять список намеченных дел и проверять статус проекта. В этом случае вы хотели бы оптимизировать приложение для как можно более быстрой загрузки JavaScript, вместо того чтобы наиболее быстро показывать содержимое пользователю.
- *Среднее (приложение для записи в блоге/сообщении)*. Пользователи должны очень быстро получать возможность читать и просматривать статьи. Это не зависит от интерактивности приложения, поэтому в данном случае оптимизируйте приложение для быстрого первого отображения.

При рассмотрении SSR стоит взвесить компромисс между использованием ресурсов при рендеринге на сервере и на клиенте. Если вы рендерите огромное количество данных (возможно, тысячи строк в онлайновой электронной таблице), выполнение на сервере, вероятно, потребует от вас отправки более крупной начальной полезной нагрузки в браузер. Это, в свою очередь, вероятно, означает более длительное TTI, которое способно нанести ущерб пользователям и, вероятно, использует больше ресурсов сервера. Получение такого же объема данных в формате JSON после загрузки приложения, например, может уменьшить полезную нагрузку и потенциально лучше удовлетворить пользователя.

Рендеринг на сервере с корпоративными и потребительскими приложениями

Вам может показаться, что обсуждение рендеринга на стороне сервера в этой главе — это что-то теоретическое, с чем вам никогда не придется иметь дело. Но, я думаю, вы обнаружите, что рендеринг на стороне сервера более распространен, чем вам кажется,

и этот вариант многие команды будут активно рассматривать. Я убедился, как это верно, на собственном опыте и опыте других разработчиков. Я трудился над потребительскими продуктами, ориентированными на широкий круг людей, и закрытыми корпоративными приложениями и имел возможность увидеть рендеринг на стороне сервера в различных бизнес-сценариях. В обоих случаях мы хотели сделать что-то лучшее для наших пользователей и рассматривали рендеринг на стороне сервера как вариант.

В корпоративном приложении мы имели дело с пользователями, которые хотели, чтобы приложение было интерактивным и быстрым, а не просто быстро рендерилось. Нам также приходилось обслуживать страницы, которые предполагалось заполнять сотнями или даже тысячами строк финансовых данных (что могло нивелировать выигрыш от рендеринга на сервере). Приложение состояло из нескольких небольших приложений, и мы обслуживали разные пакеты JavaScript в зависимости от того, какое из них использовалось в данный момент. Осложняло ситуацию то, что целостность данных и безопасность были для нас приоритетом, поэтому рендеринг на стороне сервера мог создать новую область для обеспечения безопасности и оценки с точки зрения безопасности.

Эти факторы сделали рендеринг на стороне сервера тем, что приятно иметь под рукой, что сохранится в течение некоторого времени — до тех пор, как его можно будет переоценить. Мы обнаружили, что способны сделать что-то еще, чтобы помочь пользователям, например улучшить производительность сервера, оптимизировать то, как мы обслуживаем ресурсы приложений, и отложить выборку данных на клиенте до тех пор, пока это не будет необходимо. Интересно, что люди ожидают разного от различных видов приложений. Потребительские приложения, такие как Facebook, Twitter и Amazon, борются за пользователей, которые могут выбирать из широкого спектра решений, и поэтому напрямую конкурируют между собой на многих фронтах. По моему опыту, у корпоративных пользователей несколько иные ожидания от приложения, которое они используют для работы. Скорость, конечно, невероятно важна, но также важны стабильность, надежность, понятность и другие качества бизнес-приложения. Команда разработчиков может предпочесть оптимизировать именно эти характеристики вместо того, чтобы тратить время на оптимизацию менее эффективных показателей. Это не всегда так, но так было в некоторых проектах, над которыми я работал.

В других проектах, в которых я участвовал, были очень разные требования. Одно из приложений относилось к области электронной коммерции. Рендеринг страниц на стороне сервера имел смысл, потому что время для первого отображения и соображения по SEO были чрезвычайно важны. Мы работали над тем, чтобы минимизировать размер связанных ресурсов и показывать пользователю содержимое наиболее быстро, так как любое промедление могло помешать ему делать покупки. Приложения были тесно интегрированы с работой маркетологов, поэтому обеспечение стабильной производительности SEO было приоритетной задачей.

Существуют и другие случаи, когда можно применять рендеринг на стороне сервера, но я надеюсь, что эти два простых примера помогли немного осветить некоторые практические возможности того, что мы обсуждаем в данной главе.

При реализации SSR не обязательно следовать принципу «все или ничего». Если нужно рендерить тысячи строк в электронной таблице, вероятно, имеет смысл разрешить клиенту обрабатывать такую возможность рендеринга, но при этом визуализировать страницы регистрации и авторизации на сервере, поскольку они меньше и для них важнее скорость отображения, а не интерактивность. Вы также можете рендерить определенные части страниц в Интернете, но разрешать клиенту обрабатывать все последующие извлечение и рендеринг данных. Если хотите больше узнать о различных особенностях веб-производительности, обратитесь к руководству Google по основам веб-поиска: developers.google.com/web/fundamentals/performance/.

12.3. Нужен ли вам рендеринг на стороне сервера?

Несмотря на то что SSR имеет потенциальные преимущества, следует встраивать эту технологию в свое приложение, только если это действительно необходимо. Это связано с тем, что рендеринг на стороне сервера может (в зависимости от того, насколько глубоко он интегрирован) все значительно усложнить. В этой главе мы реализуем базовую, даже упрощенную версию SSR, чтобы познакомиться с концепциями, но построение надежной специализированной реализации, которая поддерживает все нюансы SSR, может потребовать значительного количества технической работы.

Есть по крайней мере несколько причин того, что интеграция рендеринга на стороне сервера может увеличить сложность. Вот некоторые из них.

- Необходимо синхронизировать сервер и клиент таким образом, чтобы клиент мог понять, когда он берет работу на себя. Это может быть настройка разметки, обработчиков событий и многое другое, что, вероятно, понадобится клиенту. Реализация аутентификации должна также учитывать запросы, поступающие от сервера или клиента, что может потребовать изменений.
- Клиент и сервер работают в рамках различных парадигм, которые не всегда легко сопоставляются друг с другом (например, без DOM, без файловой системы и т. д.). Вы должны скоординировать передачу обслуживания и рендеринг и убедиться в том, что вы не используете, или, другими словами, правильно обрабатываете компоненты, зависящие от среды браузера.
- Хотя есть несколько исключений, React (и любой JavaScript) надежнее всего запускается во время выполнения Node.js. Это может привести к связыванию ваших клиента и сервера, который занимается рендерингом, потому что теперь им обоим необходимо поддерживать JavaScript. Возможно, это хорошо, но означает, что вы больше привязываетесь к языку/платформе JavaScript.
- Тонкая настройка SSR может потребовать специальной настройки ваших клиента и сервера. Повышение производительности обычно заключается в небольших дополнительных выигрышах, которые относятся к конкретной функциональности и почти всегда предполагают компромиссы. Иногда это означает меньшую гибкость для быстрых изменений и более сложный процесс обслуживания.

В общем, основное предупреждение таково: использовать только то, что нужно. Я не хочу, чтобы вы думали, что React-приложение не является полным или применяет какой-то неполноценный React, если оно не настроено на рендеринг на стороне сервера. Принимая решение, программисту нужно тщательно рассматривать различные компромиссы, а не только то, с чем работают другие или что популярно, этот принцип работает и здесь. Примером может служить написание вами простого приложения для ведения блога как личный проект. Реальность такова, что вам не нужны инфраструктура и технология обработки, скажем, Netflix, если вы не Netflix. Тем не менее не все крупные компании делают SSR. В настоящее время, например, даже Instagram не использует React для выполнения SSR, а эта компания много вложила в React. Применяйте то, что вам нужно.

12.4. Рендеринг компонентов на сервере

Теперь, когда мы кратко рассмотрели некоторые компромиссы рендеринга на стороне сервера, копнем глубже и посмотрим, как это работает с React. Начнем с API React, который вы будете использовать. ReactDOMServer (доступ можно получить с помощью `require('react-dom/server')` или `import ReactDOM from 'react-dom/server'`) предоставляет четыре важных метода, которые задействуются для генерации исходного кода в формате HTML для ваших компонентов:

- `renderToString`;
- `renderToStaticMarkup`;
- `renderToNodeStream`;
- `renderToStaticNodeStream`.

Давайте рассмотрим их по очереди.

У нас есть `ReactDOMServer.renderToString. renderToString`: он принимает React-элемент и генерирует соответствующую разметку HTML из компонента на основе начального состояния и свойств (по умолчанию или переданных), существующих на момент вызова метода. Элементы React, как вы помните из предыдущих разделов, являются наименьшими строительными блоками React-приложений. Они создаются с помощью `React.createElement` (или чаще всего из JSX) либо из строкового типа, либо из класса компонентов React. Метод выглядит следующим образом:

```
ReactDOMServer.renderToString(element) string
```

Когда вы выполняете рендеринг на сервере, то используете компоненты и передаете свойства как обычно. Ключевое различие между тем, к чему вы привыкли, и применением React на сервере — отсутствие инфраструктуры DOM и браузера. Это означает, что React не будет запускать методы жизненного цикла, такие как `componentWillMount` или сохраненное состояние, или использовать другие функции, специфичные для DOM.

Упражнение 12.1

Реализация сервера может оказаться довольно сложной и не должна рассматриваться как стандартная или обязательная функция для всех приложений. Потратьте некоторое время, чтобы продумать, как вы подойдете к реализации (или не станете реализовывать) рендеринга на стороне сервера для следующих типов приложений:

- корпоративного приложения без открытых для публики частей;
- сайта в социальных сетях, который сильно зависит от рекламы;
- приложения для электронной коммерции;
- платформы для видеохостинга.

`ReactDOM.renderToStaticMarkup` будет выполнять то же самое, что и `renderToString`, но без добавления каких-либо дополнительных атрибутов DOM для React для использования при захвате фокуса на стороне клиента. Это полезно для случаев, когда вы хотите выполнить базовое шаблонирование или статическое создание сайта и не нуждаетесь в каких-либо дополнительных атрибутах. `renderToStaticMarkup` почти идентичен `renderToString`:

```
ReactDOMServer.renderToStaticMarkup(element) string
```

Теперь вы не будете использовать `renderToStaticMarkup`, но, изучив, как реализовать SSR с React, станете с легкостью применять этот метод в будущих проектах там, где необходимо.

Возможно, вы заметили, что первые два метода имеют явные дополнения в `renderToNodeStream` и `renderToStaticNodeStream`. Если да, то ваша догадка верна: эти методы идентичны другим, за исключением того, что они используют API Streams узла и были введены в React 16 вместе с оптоволоконным синхронизатором и множеством других изменений. Потоки обычно применяются в node.js, и если вы с ним работали, то, вероятно, слышали о них. Если нет, можете узнать больше на nodejs.org/api/stream.html. Нам важно то, что эти потоковые методы являются асинхронными. Это дает им значительное преимущество перед синхронными аналогами. В течение некоторого времени одним из незначительных недостатков рендеринга на стороне сервера в React являлось то, что эти методы были синхронными. Это представляло проблему для приложений, которые должны рендерить сложные страницы с множеством компонентов. Мы исследуем эти методы позже, когда станем рассматривать выборку данных на сервере как части рендеринга на стороне сервера.

Теперь, когда вы узнали немного больше о доступных методах API, сосредоточимся на `renderToString`. Метод `renderToString` будет генерировать код, React может с ним работать и использовать его на клиенте. У React-DOM есть другой метод — `hydrate`, который работает почти так же, как обычный `render`, к которому вы так привыкли. Основное различие заключается в том, что `hydrate` специально обрабатывает разметку, создаваемую рендерингом на стороне сервера.

Если вы вызовете `ReactDOM.hydrate()` в узле, у которого уже есть разметка, выполненная React-DOM на сервере, React сохранит существующий HTML и будет

работать меньше, чем в противном случае. Это, как правило, означает, что у React будет еще меньше работы при первом запуске в дополнение к быстрой начальной загрузке (в зависимости от количества данных, которые вы отправляете, и других факторов, таких как загрузка сервера, сеть, погода и т. д.). Однако помните, что SSR — не магия и вы можете легко лишиться любого выигрыша в производительности, если станете загружать огромные файлы JavaScript, не будете разделять свой код или начнете противоречить другим лучшим практикам.

До сих пор мы не касались файлов на сервере. Кроме того, что объем данной главы ограничен, серверное программирование выходит за рамки темы книги, поэтому я не буду подробно описывать парадигмы программирования среды node.js или веб-сервера. Если вам интересно узнать больше о программировании в узле и на стороне сервера, ознакомьтесь с книгой «Node.js в действии»[1].

Вы начнете работу с SSR, сосредоточившись на изменениях на сервере, которые нужно сделать. В листинге 12.3 показано состояние кода сервера основного приложения до того, как он будет настроен на работу с React. Я показал код полностью, чтобы вы могли понять, как он работает. Большая часть кода — это болванка промежуточного ПО, которое может использоваться простым приложением Express, значительная его часть напрямую не связана с SSR. Рисунок 12.3 помещает код из листинга 12.3 в контекст подходов рендеринга, которые мы обсуждали до сих пор в этой главе.

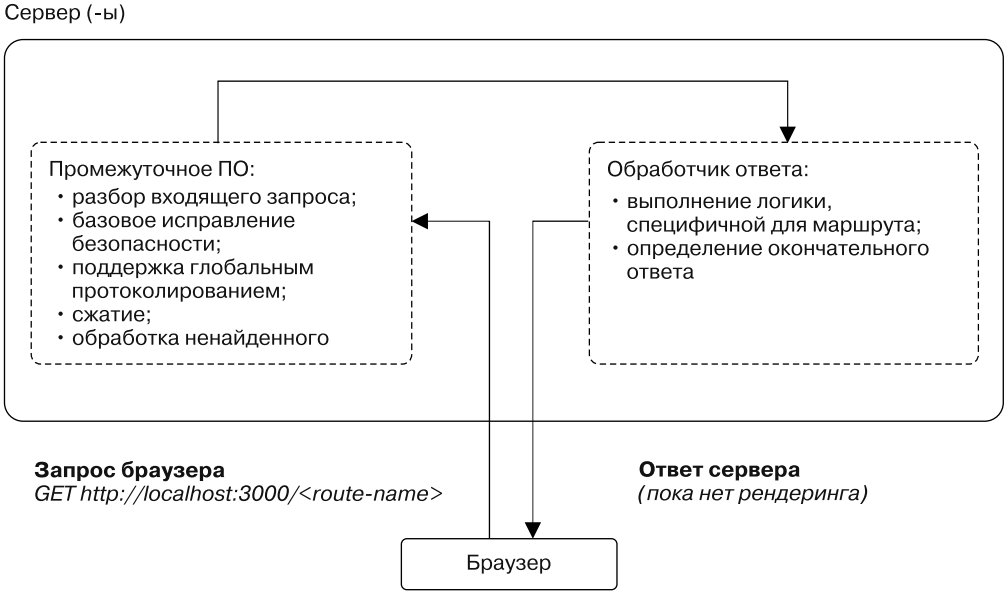


Рис. 12.3. Как и в листинге 12.3, здесь приведены основные сведения о том, что делает серверный код. Он настраивает сервер, добавляет промежуточное ПО, а затем обслуживает выделенный HTML-файл, который загружает ваше приложение

[1] *Янг А., Мек Б., Кантелон М.* Node.js в действии. 2-е изд. — СПб.: Питер, 2018.

В листинге 12.3 показана базовая настройка сервера для приложения. Когда вы поместите его в контекст подхода SSR, который рассматривается в этой главе, он будет соответствовать парадигме, ориентированной на клиента. При таком подходе сервер обычно отправляет только HTML-файл, в котором нет предварительно обработанного содержимого. Сейчас о создании и обслуживании HTML-файла заботятся инструменты сборки. Этот файл содержит ссылки на сценарии, которые будут загружаться и выполняться для реализации рендеринга и управления приложением, но на сервере рендеринг не выполнен (пока что!).

Листинг 12.3. Запуск на сервере (server/server.js)

```
import { __PRODUCTION__ } from 'environs';
import { resolve } from 'path';
import bodyParser from 'body-parser';
import compression from 'compression';
import cors from 'cors';
import express from 'express';
import helmet from 'helmet';
import favicon from 'serve-favicon';
import hpp from 'hpp';
import logger from 'morgan';
import cookieParser from 'cookie-parser';
import responseTime from 'response-time';
import * as firebase from 'firebase-admin';
import config from 'config';

import DB from '../db/DB';

const app = express();
const backend = DB();

app.use(logger(__PRODUCTION__ ? 'combined' : 'dev'));
app.use(helmet.xssFilter({ setOnOldIE: true }));
app.use(responseTime());
app.use(helmet.frameguard());
app.use(helmet.ieNoOpen());
app.use(helmet.noSniff());
app.use(helmet.hidePoweredBy({ setTo: 'react' }));
app.use(compression());
app.use(cookieParser());
app.use(bodyParser.json());
app.use(hpp());
app.use(cors({ origin: config.get('ORIGINS') }));

app.use('/api', backend);
app.use(favicon(resolve(__dirname, '..', 'static', 'assets',
  'meta', 'favicon.ico')));

app.use((req, res, next) => {
  const err = new Error('Not Found');
```

Использование синтаксиса модулей ES, доступного в узле 8.5 и выше через ESM

Настройка промежуточного ПО, которое будет применяться ко всем входящим запросам, поддерживать ведение журнала и некоторые основные меры безопасности, разбирать входящие запросы

Ответ на запросы, где вы будете интегрироваться с React DOM

```
  err.status = 404;
  next(err);
});

app.use((err, req, res) => {
  console.error(err);
  return res.status(err.status || 500).json({
    message: err.message
  });
});

module.exports = app;
```

Код обработки ошибок, который будет перехватывать переадресованные ошибки с других маршрутов и отправлять клиенту

Первый шаг, который нужно предпринять, — запустить React-DOM и попробовать рендерить простой компонент. Прежде чем перейти к интеграции своего приложения, выполните рендеринг простого `div` с текстом внутри него. В этом небольшом примере вы будете использовать `React.createElement`, поэтому заниматься преобразованием файла сервера не придется, но позже сможете применить JSX в других файлах, когда подготовите компоненты. Это потому, что вы задействуете `babel-register` — библиотеку Babel для разработки, которая преобразует код на лету. Вы можете видеть, как мы вставляем `babel-register` в index.js. В рабочей среде вы бы так не делали. Вместо этого будете использовать библиотеку типа Webpack и Babel для компиляции кода в пакет. Я не могу подробно описывать инструментарий, узнать о нем больше можно на webpack.js.org и babeljs.io.

Для первого прохода осталось только вставить простое сообщение в виде дочернего содержимого `div` и отправить его клиенту. После того как вы это сделаете, запустите сервер и проверьте, что получили в ответ. На рис. 12.4 показано, что выполняет код из листинга 12.4.

Листинг 12.4. Опробование рендеринга на стороне сервера

```
//...
app.use('/api', backend);
app.use(favicon(resolve(__dirname, '..', 'static', 'assets',
   'meta', 'favicon.ico')));
app.use('*', (req, res, next) => {
  const componentResponse = ReactDOMServer.renderToString(
    React.createElement(
      'div',
      null,
      'Rendered on the server at ${new Date()}'
    )
  );
  res.send(componentResponse).end();
});
//...
```

В обработчике запросов создается строка HTML, а затем отправляется

Использование renderToString и передача каркаса React-элемента

Создание элемента div без свойств

Передача простой строки с отметкой времени как дочернего содержимого

Отправка ответа клиенту

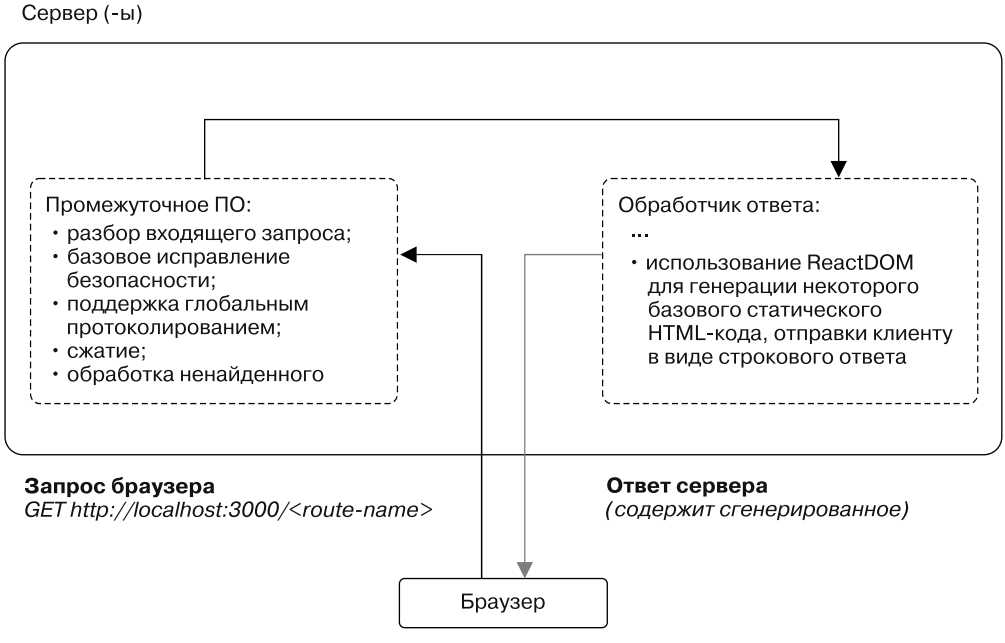


Рис. 12.4. Теперь вы используете React-DOM для рендеринга простой строки HTML и ее отправки клиенту. В некотором смысле это все, чем является SSR (выполнить статическую разметку, отправить ее клиенту). Сложность, о которой я говорил, как правило, в получении всех данных, необходимых для создания текста, координации процесса с клиентом и дальнейшей оптимизации

Если вы внесете изменение в листинг 12.4, запустите сервер с помощью команды `node server/run.js` в терминале и примените другой сеанс для отправки запроса с помощью cURL, то увидите ответ, возвращенный сервером. Раньше вы отправляли одну и ту же строку HTML каждый раз, и этот документ должен был загружать ваши сценарии приложения. Затем React запускал и рендерил приложение в DOM (создавая узлы DOM, назначая слушателей событий и т. д.). С помощью нового подхода вы можете делегировать начальный рендеринг серверу и позволить React взять инициативу на себя. В листинге 12.5 показано, как запустить сервер и использовать cURL для проверки ответов, приходящих с него.

Листинг 12.5. Проверка первого ответа, подготовленного сервером

```
$ npm run server:dev

// ... in a different terminal session

$ curl -v http://localhost:3000
> GET / HTTP/1.1
> Host: localhost:3000
> User-Agent: curl/7.51.0
```

Отправка запроса серверу, проверка ответа

```
> Accept: */*
>
< HTTP/1.1 200 OK
< X-Powered-By: react
< X-XSS-Protection: 1; mode=block
< X-Frame-Options: SAMEORIGIN
< X-Download-Options: noopen
< X-Content-Type-Options: nosniff
< Access-Control-Allow-Origin: *
< Content-Type: text/html; charset=utf-8
< Content-Length: 144
< ETag: W/"90-gXhNJUy73fc2MSrpr7eaKDZ7OV8"
< Vary: Accept-Encoding
< X-Response-Time: 0.795ms
< Date: Mon, 08 May 2017 10:26:55 GMT
< Connection: keep-alive
<
* Curl_http_done: called premature == 0
* Connection #0 to host localhost left intact

<div data-reactroot="">Rendered on the server at Mon May 08 2017 03:26:55
  GMT-0700 (PDT)</div>
```

Вы должны получить заголовки обратно в своем запросе, но тело ответа важнее

Специальные свойства react-root и react-checksum во внешнем HTML-элементе

Рендеринг на стороне сервера выполнен. Вы задействовали React для создания строкового представления компонента React и отправки его клиенту. Прямо сейчас React не загружен, поэтому не подхватит обработку с того места, где остановился сервер, но, как только он будет включен, сможет взять ее на себя. Попробуйте выполнить те же команды, но теперь используйте `renderToStaticMarkup` и посмотрите, чем отличается HTTP-отклик от отклика сервера.

12.5. Переход на React Router

Роутер, созданный в предыдущих главах, оптимизирован для обработки маршрутизации в браузере, но разработан без учета рендеринга на стороне сервера. Возможность вникнуть и посмотреть, что реально выполнить с помощью React, — важный результат создания роутера, а не просто установка сторонней библиотеки, и я надеюсь, вы увидели, как компоненты могут использоваться по-другому.

Ваш роутер может быть полезен при относительно простых потребностях приложения, рассматриваемого в качестве примера, но недостаточно функционален в нескольких областях. У него довольно простой API, и было бы неплохо, если бы он поддерживал такие функции, как перехватчики маршрутизации (переходы между маршрутами), промежуточное ПО (логика, которая применяется к нескольким маршрутам), и многое другое. И по мере углубления в рендеринг на стороне сервера с помощью React вам понадобится большая функциональность, например возможность генерировать дерево компонентов для рендеринга на

основе URL-адреса запроса. Вот почему вы будете работать с библиотекой React Router V3.

Библиотека React Router (github.com/ReactTraining/react-router), по-видимому, является единственным наиболее широко используемым и доработанным решением маршрутизации для React. Ей обеспечено надежное сопровождение, разработкой занято сообщество GitHub, она претерпела несколько крупных переделок.

На момент написания книги последняя версия React Router — четвертая. Сейчас она находится в процессе тестирования и к моменту, когда вы будете читать книгу, может быть заменена основной версией. Вы возьмете третью версию, потому что ее API похож на созданный вами роутер, так что изменения потребуются небольшие. А еще потому, что это надежная технология, разработанная сообществом React с открытым исходным кодом. Она способна на большее, чем простой роутер, и даже превосходит ваши потребности.

Стоит отметить, что React Router — это целая технология, и здесь мы очень неглубоко окунемся в ее потенциал. Проект обладает множеством свойств, связанных с множеством функций маршрутизации. Последняя крупная версия (четвертая на момент написания) даже имеет решения для маршрутизации с платформой React Native. Количество разработчиков, применяющих React Router, помогло сделать проект невероятно полезным, но у него есть и недостаток — иногда он существенно меняется от предыдущей основной версии к следующей. Именно по этой причине и из-за сходства с роутером, который создали с нуля, вы не будете использовать последнюю версию React Router. Если же захотите это сделать, в моем блоге есть запись, посвященная React Router v4 с React 16: ifelse.io/2017/09/07/server-rendering-with-react-router-andreact-16-fiber. Я также отмечу следующее: хотя между версиями React Router API изменились, к ним применимо большинство прежних концепций — просто при переходе нужно переназначить функциональность новым API.

Выбор сторонних библиотек против создания собственных

Еще одна причина, по которой вы переходите на React Router вместо того, чтобы придерживаться доморощенного решения, заключается в том, что это более вероятный кандидат для применения в любых деловых ситуациях, в которых окажетесь вы или ваша команда. Можете выбрать решение с открытым исходным кодом, например React Router, вместо написания собственного. Это вызвано тем, что потребности обусловливают, окажется ли целесообразным тратить на это время, необходимое для создания и поддержания надежного решения проблемы. Решить, покупать или разрабатывать, может быть сложно, когда дело доходит до внешних зависимостей. Мой совет здесь — учитывать два момента:

- не нужно искать что-то еще, потому что все остальные работают с этой библиотекой;
- часто намного сложнее разработать собственное решение, а на поддержку вообще затрачивается больше всего времени.

К тому же большое сообщество разработчиков приложений с открытым исходным кодом способно обнаружить множество ошибок, прежде чем вы столкнетесь с ними.

Настройка роутера React. Мы решили использовать библиотеку React Router в качестве готовой замены для вашего роутера, поэтому рассмотрим ее настройку. Первый шаг — убедиться, что взамен текущего роутера установлена библиотека React Router. Несмотря на то что технологии различаются, применяемые API будут схожи.

Библиотека React Router должна устанавливаться, уже обладая зависимостями проекта. Теперь нужно перевести свой проект на React Router и настроить его так, чтобы выполнялся рендеринг на стороне сервера. Начните с файла `src/index.js`. Это файл начальной настройки, в котором вы настраивали основные части приложения, включая прослушивание истории браузера, рендеринг компонента роутера и активизацию слушателя событий аутентификации.

Предыдущая настройка рендеринга на стороне сервера не подойдет, потому что бóльшая часть кода зависит от среды браузера, а вам не нужны все функции React Router, чтобы приложение смогло работать. Все, что нужно сохранить, — это слушатель аутентификации. Прежде чем добавить функционал, создайте вспомогательный инструмент для дальнейшей работы. В листинге 12.6 показано, как написать простую утилиту для проверки нахождения в среде браузера. Некоторые технологии, такие как Webpack, могут связать код, который нужен для среды, но для наших целей сгодится и более простой подход.

Листинг 12.6. Проверка среды браузера (src/utils/environment.js)

```
export function isServer() {
  return typeof window === 'undefined';
}
```

Теперь можете задействовать этот помощник, чтобы определить, в какой среде находитесь, и выполнять код условно, в зависимости от ваших потребностей. Он не выполняет исчерпывающие проверки, чтобы убедиться, что вы находитесь в среде браузера, но для ваших нужд этого достаточно. Необходимо учитывать среду, в которой работает код, — довольно распространенный показатель создания приложений с возможностями рендеринга на стороне сервера или приложений, совместно использующих код между клиентом и сервером (иногда называемый *универсальным* или *изоморфным*). По моему опыту, это часто становится источником ошибок, которые трудно отследить, особенно если вы устанавливаете сторонние зависимости, выполненные без учета окружения.

К настоящему времени многие из существующих в сообществе React технологий либо поддерживают рендеринг на стороне сервера, либо указывают, что может вызвать проблемы. Но так бывает не всегда. Работая с более ранними версиями React несколько лет назад, я столкнулся с ошибками в самой React, которые сделали некоторые функции определенных библиотек непредсказуемыми. Сейчас все намного лучше, и рендерингом на стороне сервера занимается не только сообщество React, но и основная команда.

Прежде чем читать дальше, вам нужно внести незначительные исправления в один из редукторов, чтобы учитывать окружение сервера. Пользовательский

редуктор установит cookie в браузере с помощью `js-cookie`. Сервер обычно не позволяет хранить cookie (хотя есть библиотеки, которые могут эмулировать это поведение, например `tough-cookie` (github.com/salesforce/tough-cookie)), поэтому нужно использовать помощник для настройки данного кода. В листинге 12.7 перечислены необходимые изменения.

Листинг 12.7. Модификация пользовательского редуктора

```
export function user(state = initialState.user, action) {
  switch (action.type) {
    case types.auth.LOGIN_SUCCESS:
      const { user, token } = action;
      if (!isServer()) {                                  Попытка использовать cookie браузера,
        Cookies.set('letters-token', token);              если вы находитесь в среде браузера
      }
      return Object.assign({}, state.user, {
        authenticated: true,
        name: user.name,
        id: user.id,
        profilePicture: user.profilePicture ||
          '/static/assets/users/4.jpeg',
        token
      });
    case types.auth.LOGOUT_SUCCESS:
      Cookies.remove('letters-token');
      return initialState.user;
    default:
      return state;
  }
}
```

Вернемся к задаче. Вам необходимо настроить React Router. Подобно роутеру, библиотека React Router (третья версия) позволяет применять вложенную иерархию компонентов `<Route/>`, чтобы указать, какие из них должны быть сопоставлены с какими URL-адресами. Как я уже отмечал, React Router — это невероятно широко используемое и проверенное временем решение с множеством функций, которые не придется добавлять к роутеру, — вы просто замените им свой роутер и не будете изучать все, что он может делать.

Создайте файл `src/routes.js` для своих маршрутов. Вы выделяете для маршрутов собственный файл, потому что нужно получить доступ к нему с сервера и клиента. Это удобно для приложений, где клиентский код находится рядом с кодом сервера, но вам может потребоваться найти другой способ загрузки маршрутов на сервер, если они размещены в другом месте (через npm, подмодуль Git и т. д.). Файл маршрутов должен выглядеть как выполненный вами роутер — с несколькими незначительными отличиями. Вы добавили возможность указать компонент индекса в том же компоненте `<Route/>`, в то время как React Router предоставляет для этой цели отдельный компонент. На рис. 12.5 схематично показана роль конфигурации маршрутов. Она работает в том же общем виде, что и роутер, и служит для

Рис. 12.5. Так же как и построенный роутер, конфигурация маршрутов для React Router сопоставляет URL-адреса с компонентами. Вы можете вкладывать компоненты для совместного использования некоторых частей пользовательского интерфейса разными страницами или подразделами (например, панель навигации или другой общий компонент)

сопоставления URL-адресов с компонентами или деревьями компонентов (в случае вложенности). В листинге 12.8 показано, как интегрировать React Router в настройку маршрутизации.

Листинг 12.8. Создание маршрутов для React Router (src/routes.js)

```
import React from 'react';

import App from './pages/app';
import Home from './pages/index';
import SinglePost from './pages/post';
import Login from './pages/login';
import NotFound from './pages/404';

import { Route, IndexRoute } from 'react-router';

export const routes = (
  <Route path="/" component={App}>
    <IndexRoute component={Home} />
    <Route path="posts/:post" component={SinglePost} />
    <Route path="login" component={Login} />
    <Route path="*" component={NotFound} />
  </Route>
);
```

Использование App, чтобы обернуть все приложение

Использование компонента IndexRoute React Router, чтобы показывать компоненты по индексным путям (/)

Сопоставление компонентов с путями, как было с вашим роутером

Теперь, когда у вас есть какие-то маршруты, можете импортировать их в основной файл приложения для использования с React Router. Те же маршруты будут применяться на клиенте и сервере, с участием *универсального* или *изоморфного* аспекта рендеринга на стороне сервера, о котором вы, вероятно, слышали. Повторное использование кода на клиенте и сервере может иметь большое значение, но в нашем локальном случае вы, вероятно, не увидите значительной выгоды. Преимущество, которое вы получите, заключается в том, что вы легко покажете свои клиентские компоненты серверу «нормальным» способом React.

Теперь импортируйте маршруты в свой сервер. В листинге 12.9 показано, как доставить маршруты на сервер и задействовать их в процессе рендеринга. Как сервер собирается захватить нужный для рендеринга компонент (компоненты)? Поскольку маршрутизация — это лишь сопоставление URL-адресов с действиями (в данном случае ответами HTTP), вы должны найти правильный соответствующий компонент. В своем роутере вы применяли базовую библиотеку соответствия URL-адресов регулярным выражениям, чтобы определить, был ли URL-адрес сопоставлен с компонентом роутера. Она определила, какой компонент, если таковой имеется, должен быть представлен на основе URL-адреса (см. рис. 12.5). React Router позволит делать то же самое, но на сервере. Таким образом, можно использовать URL-адрес из входящего на сервер HTTP-запроса, чтобы он соответствовал компоненту (компонентам) для рендеринга в статическую разметку. Это ключевая точка соприкосновения между React Router и стоящей перед вами целью использования SSR. React Router применяет URL-адрес для рендеринга компонентов или дерева

компонентов как обычно, но на сервере. В листинге 12.9 показано, как настроить исходную серверную часть SSR с помощью React Router.

Листинг 12.9. Использование React Router на сервере (server/server.js)

```
//...
import { renderToString } from 'react-dom/server';
import React from 'react';
import { match, RouterContext } from 'react-router';
import { Provider } from 'react-redux';

import configureStore from '../src/store/configureStore';
import initialReduxState from '../src/constants/initialState';
import { routes } from '../src/routes';
//...
app.use('*', (req, res) => {
  match({ routes: routes, location: req.originalUrl },
  (err, redirectLocation, props) => {
    if (redirectLocation && req.originalUrl !== '/login') {
      return res.redirect(302, redirectLocation.pathname +
      redirectLocation.search);
    }

    const store = configureStore(initialReduxState);
    const appHtml = renderToString(
      <Provider store={store}>
        <RouterContext {...props} />
      </Provider>
    );

    const html = '
      <!doctype html>
      <html>
        <head>
          <link rel="stylesheet"
            href="http://localhost:3100/static/styles.css" />
          <meta charset=utf-8/>
          <meta http-equiv="X-UA-Compatible" content="IE=edge">
          <title>Letters Social | React In Action
             by Mark Thomas</title>
          <meta name="viewport" content="width=devicewidth,
            initial-scale=1">
        </head>
        <body>
          <div id="app">
            ${appHtml}
          </div>
        <script src="http://localhost:3000/bundle.js"
          type='text/javascript'></script>
```

Передача URL-адреса в функцию match, а также маршрутов

Импортирование некоторых утилит из React Router, renderToString из React DOM, компонента Redux Provider, хранилища и маршрутов

match выдает ошибку, перенаправляет ее (если она есть) и свойства. Будет использоваться для рендеринга пользовательской страницы ошибки или перенаправления

Передача компонента RouterContext, который вы импортировали из React Router, и оборачивание его в обычный компонент Redux Provider

Использование литерала строкового шаблона для создания HTML-документа с добавленным в него HTML-кодом приложения

```
          </body>
        </html>
      '.trim();
      res.setHeader('Content-type', 'text/html');
      res.send(html).end();
    });
});

//... Ошибка обработки

export default app;
```

Установка заголовков в ответе и отправка обратно в браузер

12.6. Обработка аутентифицированных маршрутов с помощью роутера React

Теперь, когда вы настроили сервер, можете немного почистить клиентскую часть своего приложения. Вам нужно убедиться, что задействуется новая настройка маршрутизации. Также необходимо переместить часть выстроенной логики, связанной с аутентификацией, чтобы лучше использовать React Router. Для этого вы станете работать с набором функций, доступных в React Router, — подключениями к жизненному циклу. Подобно тому как методы жизненного цикла работают для монтирования, обновления и размонтирования компонентов, React Router предоставляет определенные перехватчики жизненного цикла для переходов между маршрутами. Существует несколько способов их применения.

- Вы можете инициировать сбор данных для страницы или проверить, зарегистрирован ли пользователь, прежде чем разрешить ему завершить переход по URL.
- Можете поддерживать любую очистку или завершать аналитический сеанс, когда пользователь покидает страницу, — вы не ограничены событиями, связанными с вводом.
- С помощью перехватчиков жизненного цикла React Router возможно выполнять синхронную *или* асинхронную работу, поэтому вы не ограничены ни одной из них.
- Можете отправлять события просмотра страниц на платформу аналитики, такую как Google Analytics.

На рис. 12.6 показан базовый поток перехватчиков жизненного цикла, применяемых в React Router v3. React Router взаимодействует с API History (developer.mozilla.org/ru/docs/Web/API/History_API) за кадром, но предоставляет эти перехватчики для упрощения маршрутизации в приложениях. Если вы хотите узнать больше о React Router v3 API и изучить другие полезные руководства, написанные участниками сообщества, прочитайте документацию на сайте GitHub по адресу github.com/ReactTraining/react-router/blob/v3/ Docs/API.md.

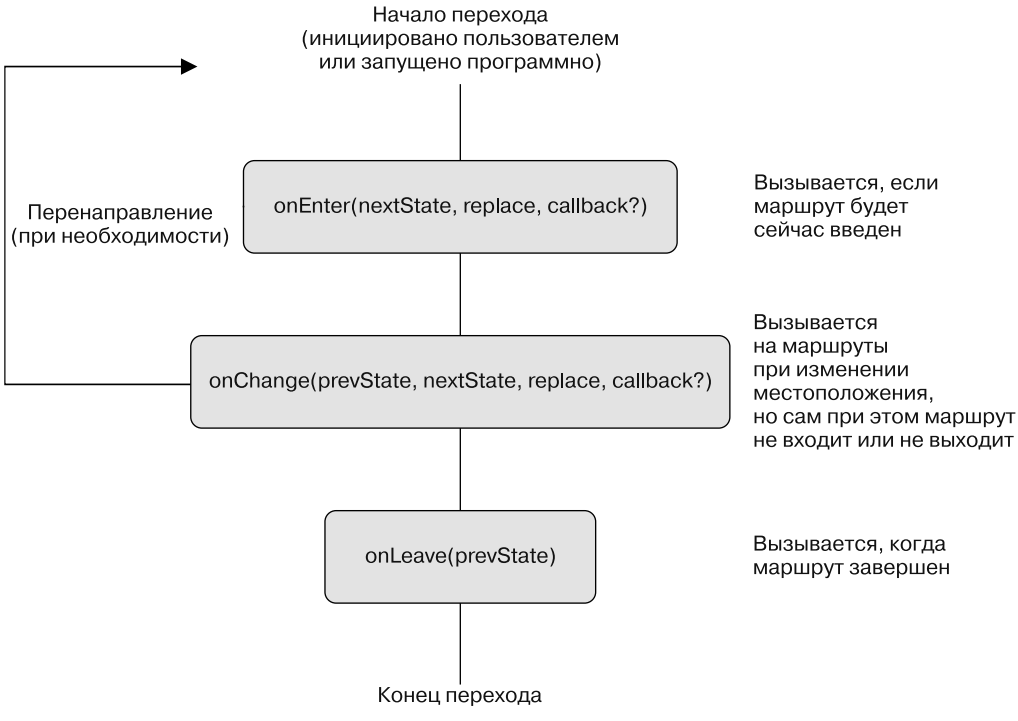


Рис. 12.6. React Router предоставляет несколько обработчиков событий для компонентов Route. Вы можете с их помощью перехватывать маршруты, которые возникают, когда пользователь или код вызывает переход. Обратите внимание на то, что перенаправление *не* является перенаправлением HTTP с кодом состояния 3XX

Вы будете использовать перехватчики жизненного цикла `onEnter` для проверки авторизации пользователя для определенных маршрутов и перенаправления на страницу входа в систему, если аутентифицированного пользователя нет. На практике вы хотели бы рассмотреть свое приложение с точки зрения безопасности и потратить некоторое время на то, чтобы не позволить пользователю переходить на страницы, куда он попасть не должен. Необходимо также обеспечить распространение вашей стратегии безопасности и на сервер. Но сейчас для защиты некоторых маршрутов должно быть достаточно Firebase и перехватов маршрутов. В листинге 12.10 показано, как настроить перехватчики жизненного цикла `onEnter` для защищенных страниц. Вы можете узнать о логике аутентификации из главы 11, где она использовалась в ходе авторизации. На рис. 12.6 показано, как работает этот процесс.

Финальный прием настройки, который вам нужно применить, прежде чем читать дальше, — очистить основной файл приложения и заменить компоненты ссылок. В листинге 12.11 показана урезанная версия основного файла на стороне клиента.

Листинг 12.10. Настройка перехватчика жизненного цикла onEnter (src/routes.js)

```
import React from 'react';

import { Route, IndexRoute } from 'react-router';

import App from './pages/app';
import Home from './pages/index';
import SinglePost from './pages/post';
import Login from './pages/login';
import Profile from './pages/profile';
import NotFound from './pages/error';
import { firebase } from './backend';
import { isServer } from './utils/environment';
import { getFirebaseUser, getFirebaseToken } from './backend/auth';

async function requireUser(nextState, replace, callback) {
  if (isServer()) {
    return callback();
  }
  try {
    const isOnLoginPage = nextState.location.pathname === '/login';

    const firebaseUser = await getFirebaseUser();
    const fireBaseToken = await getFirebaseToken();

    const noUser = !firebaseUser || !fireBaseToken;

    if (noUser && !isOnLoginPage && !isServer()) {
      replace({
        pathname: '/login'
      });
      return callback();
    }
    if (noUser && isOnLoginPage) {
      return callback();
    }
    return callback();
  } catch (err) {
    return callback(err);
  }
}

export const routes = (
  <Route path="/" component={App}>
    <IndexRoute component={Home} onEnter={requireUser} />
    <Route path="/posts/:postId" component={SinglePost}
      onEnter={requireUser} />
    <Route path="/login" component={Login} />
    <Route path="*" component={NotFound} />
  </Route>
);
```

Перехваты React Router принимают три аргумента: nextState, функцию замены и обратный вызов

Импортирование утилиты Firebase и isServer

Если находится на сервере, продолжайте

Вам нужно знать, находитесь ли вы на странице авторизации, чтобы не перенаправлять бесконечно

Использование служебных функций Firebase, включенных в репозиторий примеров, чтобы получить пользователя и токен Firebase

Если нет токена или пользователя и вы не на странице авторизации, перенаправление пользователя

Если нет пользователя, но он находится на странице авторизации, ему разрешается продолжить

Если ошибка — обратный вызов с ней

Добавление подключения к жизненному циклу к соответствующим компонентам с помощью свойства

Листинг 12.11. Очистка индексов приложения (src/index.js)


```
import React from 'react';
import { hydrate } from 'react-dom';
import { Provider } from 'react-redux';

import { Router, browserHistory } from 'react-router';
import configureStore from './store/configureStore';
import initialReduxState from './constants/initialState';
import { routes } from './routes';

import './shared/crash';
import './shared/service-worker';
import './shared/vendor';
// Примечание: это не очень согласуется с ES*,
// но работает, так как мы используем
  Webpack as a build tool
import './styles/styles.scss';

// Создание хранилища Redux
const store = configureStore(initialReduxState);

hydrate(
  <Provider store={store}>
    <Router history={browserHistory} routes={routes} />
  </Provider>,
  document.getElementById('app')
);
```

Вы настроили React Router с помощью `browserHistory`, но можете также настроить эту библиотеку либо с помощью истории из хеша, либо из памяти. Они немного отличаются от истории браузера тем, что не используют тот же API History браузера. История на основе хеша работает, изменяя хешированный фрагмент в URL-адресе, но оставляя неизменной историю браузера пользователя. API истории в памяти не манипулирует URL-адресом вообще и лучше подходит для таких ситуаций, как локальная разработка или React Native (рассматривается в следующей главе). Для получения дополнительной информации о доступных реализациях истории см. github.com/ReactTraining/react-router/blob/v3/docs/guides/Histories.md.

Если вы запускаете приложение локально, то увидите все, что будет отображено на сервере и отправлено клиенту. React должна принять руководство процессом на себя, и все должно быть интерактивным, как и следовало ожидать. Однако можно заметить, что маршрутизация со ссылками, кажется, нарушена. Это потому, что вы создали собственные компоненты со ссылками, которые интегрируются со старым роутером. К счастью, для решения этой проблемы нужно всего лишь заменить использованный модуль тем, который применяет React Router. Переключиться просто, но стоит отметить, что выбор или написание роутера способны повлиять на значительную часть приложения. Ссылки, переключающие между страницами, способы доступа к свойствам — все это может быть затронуто маршрутизацией, и вы должны это учесть.

Главное изменение, которое требуется сделать, — это замена истории ваших ссылок. React Router по-прежнему использует API History браузера, но вы можете

синхронизировать его с роутером, применив React Router. Поскольку вы централизовали обертку навигации, любые действия, необходимые для того, чтобы направлять пользователей, должны отлично работать в рамках новой настройки. В листинге 12.12 показаны строки, которые нужно изменить. И больше ничего.

Листинг 12.12. Переключение истории (src/history/history.js)

```
import { browserHistory } from 'react-router';
const history = typeof window !== 'undefined'
  ? browserHistory
  : { push: () => {} };
const navigate = to => history.push(to);
export { history, navigate };
```

Единственные строки, которые вам нужно изменить. Сообщите React Router о сделанных переходах

Вы должны выполнять рендеринг на сервере с помощью React Router с учетом этих изменений! Повторим, что происходит.

- Когда приходит запрос, вы передаете его URL-адрес утилите `match` React Router, чтобы получить компонент (компоненты), который хотите отобразить.
- Используя результаты из `match`, вы применяете метод `renderToString` React DOM для формирования ответа HTML и отправки его обратно клиенту.
- Если вы задействуете cURL или инструменты разработчика для проверки сервера разработки, запущенного с помощью `npm run server:dev`, то должны увидеть HTML для своих компонентов в ответе (рис. 12.7).

Рис. 12.7. Проверка приложения с рендерингом на стороне сервера. С помощью React-DOM можно создать HTML-код приложения и отправить его клиенту. Обратите внимание на то, что, поскольку вы не сделали выборки данных на стороне сервера, не ожидайте увидеть какие-либо динамические данные, содержащиеся в приложении (например, сообщения)

12.7. Рендеринг на стороне сервера с получением данных

Вы интегрировали в свое приложение рендеринг на стороне сервера. Сейчас могут проявиться преимущества в сфере вызова приложения и производительности. Однако есть возможность улучшить ситуацию. В данный момент вы не делаете ничего, чтобы рендерить завершенное приложение перед его отсылкой. Полезная нагрузка, которую отправляете, одинакова независимо от того, зарегистрирован пользователь или нет. На браузер возложены такие действия, как запуск потока аутентификации и загрузка сообщений. Реализация сервера также является синхронной, поскольку вы еще не используете `renderToNodeStream`. В данном разделе вы улучшите рендеринг на стороне сервера, чтобы воспользоваться преимуществом этого API и интегрировать Firebase на сервер, чтобы можно было рендерить, зная о состоянии аутентификации. На рис. 12.8 показан общий процесс рендеринга на стороне сервера с интегрированной выборкой данных.

Firebase обеспечивает способ взаимодействия с API для сервера аналогично тому, как для браузера. Так вы сможете рассматривать Firebase как базу данных на сервере. В других ситуациях возможно выполнить что-то вроде HTTP-вызова микросервиса или базы данных, чтобы определить, существует ли пользователь и аутентифицирован ли он. Вы будете придерживаться Firebase, потому что сфокусированы на React, но обратите внимание: можете загрузить одну из этих систем при разных обстоятельствах.

Если вы еще не создали учетную запись Firebase, сейчас самое время. Я выложил исходники приложения с общедоступным токеном для учетной записи, но для применения API-интерфейсов пользователей Firebase нужна реальная учетная запись (можете использовать ее для доступа к информации о пользователях, чего я не хочу). Чтобы настроить учетную запись Firebase, перейдите на страницу `firebase.google.com` и зарегистрируйтесь для получения учетной записи (или возьмите существующую учетную запись Google). На сайте создайте проект с названием, которое вам нравится.

После этого следует настроить SDK администратора Firebase. Этот процесс может быть изменен в новых версиях, поэтому здесь я не буду рассказывать о нем. Инструкции по установке и настройке находятся на странице `firebase.google.com/docs/admin/setup`, они довольно просты. Мы больше всего нуждаемся в API User Management. Вам не нужно устанавливать что-либо еще в проекте, потому что SDK node.js Firebase уже включен в его зависимости.

Завершая настройку, замените ключи Firebase, включенные в приложение, так как они связаны с проектом Letters Social и, скорее всего, будут конфликтовать с вашими. Вы найдете их в исходном коде, заглянув в каталог конфигурации. Два файла, `development.json` и `production.json`, содержат переменные конфигурации для среды разработки и рабочей среды соответственно. Не бойтесь редактировать переменные по своему усмотрению (возможно, вы хотите настроить приложение

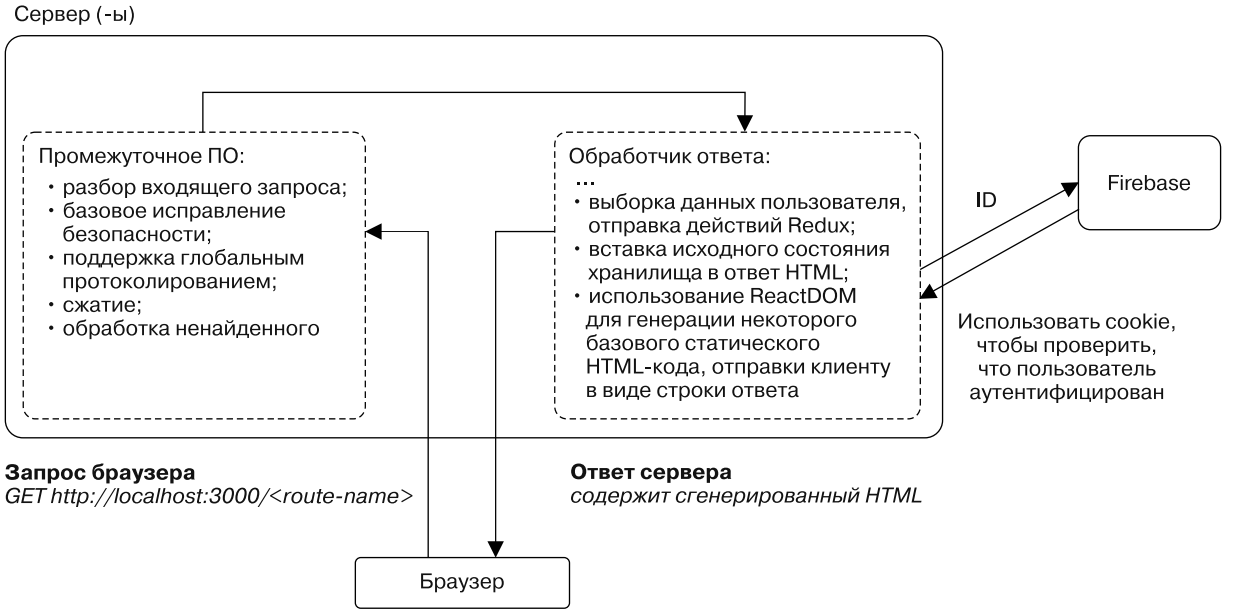


Рис. 12.8. Реализация сервера с извлечением данных. В целом это похоже на процесс рендеринга, основное отличие — нужно выполнить извлечение данных как часть рендеринга. Результат рендеринга будет меняться в зависимости от того, авторизовался пользователь или нет, как выглядят его данные и когда он авторизовался

самостоятельно и развернуть его на сайте!). На рис. 12.9 показаны консоль Firebase и страница служебной учетной записи. Сгенерируйте новый закрытый ключ и переместите загруженный файл в основной репозиторий приложения — вскоре он вам потребуется.

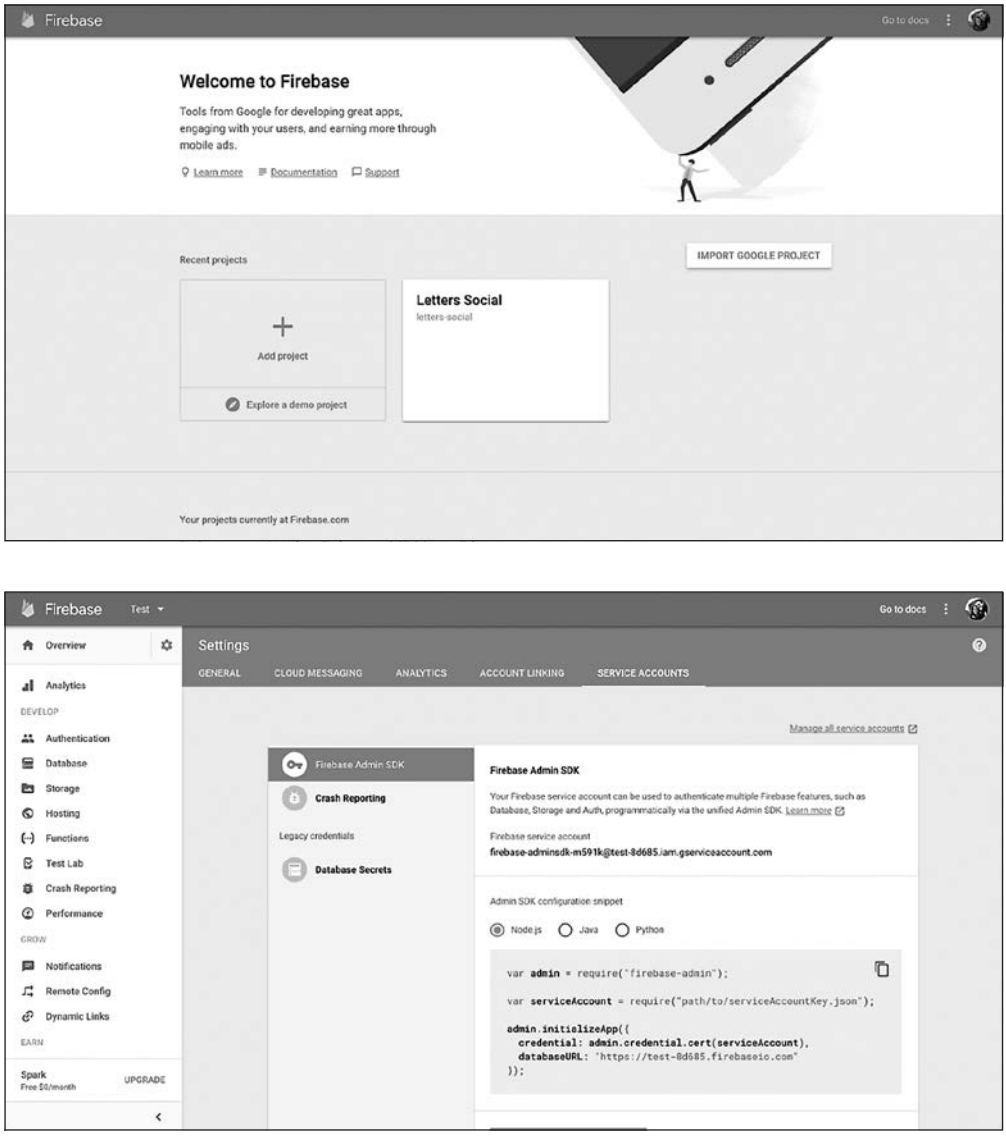


Рис. 12.9. Создайте новый проект Firebase и сгенерируйте закрытый ключ. Это позволит пройти аутентификацию на платформе Firebase и использовать SDK для управления пользователями на сервере

Разобравшись с логистикой, можете вернуться к кодированию. Вам нужно аутентифицировать свое серверное приложение с помощью платформы Firebase, чтобы проверять и выбирать пользователей Firebase для рендеринга полного состояния приложения. Вероятно, вы уже видели фрагмент примера, показывающий, как это сделать, на странице Firebase, но в листинге 12.13 показано, как настроить SDK Firebase Admin именно на вашем сервере.

Листинг 12.13. Интеграция Firebase на сервере (server/server.js)

```
// ...
import * as firebase from 'firebase-admin';
import config from 'config';

// Инициализация Firebase
firebase.initializeApp({
  credential: firebase.credential.cert(JSON.parse
    (process.env.LETTERS_FIREBASE_ADMIN_KEY) ),
  databaseURL: 'https://letters-social.firebaseio.com'
});

// const serviceAccount = require("path/to/serviceAccountKey.json");
// admin.initializeApp({
// credential: firebase.credential.cert(serviceAccount),
// databaseURL: "https://test-8d685.firebaseio.com"
// });

// Фиктивная база данных клиентов
import DB from '../db/DB';

//...
```

Импортирование SDK firebase-admin

Установка строковой версии JSON-файла в качестве переменной окружения. Разберите его, чтобы мог работать Firebase

Другой способ аутентификации с Firebase

Теперь, когда сервер запущен, он автоматически подключится к Firebase и позволит использовать SDK администратора для взаимодействия с пользователями. Таким образом, вы сможете делать выборку данных на сервере, чтобы узнать о пользователе, отправляющем запрос. Почему это важно? Помните, ранее в этой главе я упоминал, что маршрутизация на стороне сервера способна оказаться непростой, поскольку может включать синхронизацию клиента и сервера? Вы не собираетесь делать ничего ужасно сложного, но это как раз то, о чем я говорил. Обработка на стороне сервера очень скоро может чрезвычайно усложниться.

К счастью, вы не станете делать ничего настолько сложного. Всего лишь используете Redux новым способом. Поскольку в Redux нет ничего, что бы ограничивало его работу в браузере, задействуйте его для управления состоянием на сервере. Далее приведен краткий обзор действий для выполнения рендеринга и получения данных.

- Получить токен пользователя из cookie, который вы сохранили в предыдущих главах.
- Проверить токен с помощью Firebase и выбрать пользователя, если он существует.

- Если у него нет действительного токена (возможно, он просрочен), очистить cookie и отправить пользователя на страницу авторизации.
- Если это действительный пользователь, извлечь его информацию с сервера и отправить действия в хранилище.
- Выполнить рендеринг соответствующего компонента маршрута на основе состояния хранилища.
- Выполнить `JSON.stringify` текущего состояния хранилища и вставить его в HTML, который необходимо отправить в браузер.

Кажется, что это очень сложно, но не пугайтесь. Вы добавляете незначительный шаг к потоку рендеринга на стороне сервера, который создали раньше. Вместо того чтобы каждый раз рендерить одно и то же содержимое, вы извлекаете данные из Firebase и используете эту информацию для рендеринга. Помните: преимущество заключается в том, что вы можете рендерить приложение целиком, чтобы пользователь мог немедленно увидеть содержимое.

Применение Redux на сервере — отличный пример универсального JavaScript в действии. Если Redux сильно зависит от API браузера, может быть сложно или невозможно интегрировать его на сервере и вам придется использовать другой подход. Тем не менее реально повторно создать хранилище по требованию, обновить его на основе ответов от API и Firebase, а затем задействовать для рендеринга приложения так же, как в браузере. На рис. 12.10 показан этот процесс в контексте рендеринга на сервере, который мы рассматривали ранее.

В этом потоке вы задействуете cookie, приходящий из браузера, чтобы убедиться, что токен пользователя действителен. Затем получаете пользователя из Firebase и отправляете действия в хранилище Redux, созданное на стороне сервера. Вы попрежнему рендерите в статический HTML, но на этот раз используете обновленное состояние, чтобы приложение можно было рендерить с новыми данными. А также вставляете состояние в ответ HTML, чтобы браузер мог забрать его, когда остановится сервер. Помните лишь, что хранилище Redux не воссоздается и не сохраняется в памяти на сервере. Я работал над проектами, где в ходе локальной разработки это происходило, что было трудно отследить. Это не только вызывало раздражение, но и означало, что сервер будет рендерить одни и те же пользовательские данные для всех, кто делает запросы, потому что состояние хранилища не было уничтожено. Это было бы неприемлемой угрозой безопасности рабочей среды. Я упоминаю об этом, чтобы помочь вернуться в реальность, где координация браузера и клиента может оказаться сложной и должна быть выполнена тщательно, чтобы избежать нестандартных ошибок или уязвимостей для системы безопасности.

Рассмотрим код, необходимый для извлечения и рендеринга данных. В листинге 12.14 показаны начальные этапы сбора данных и обработки некоторых основных ошибок, которые могут возникнуть в результате появления истекшего или недействительного токена. На следующем шаге вы интегрируете асинхронный рендеринг на стороне сервера с помощью `renderToNodeStream` из React-DOM и даже улучшите рендеринг на стороне сервера.

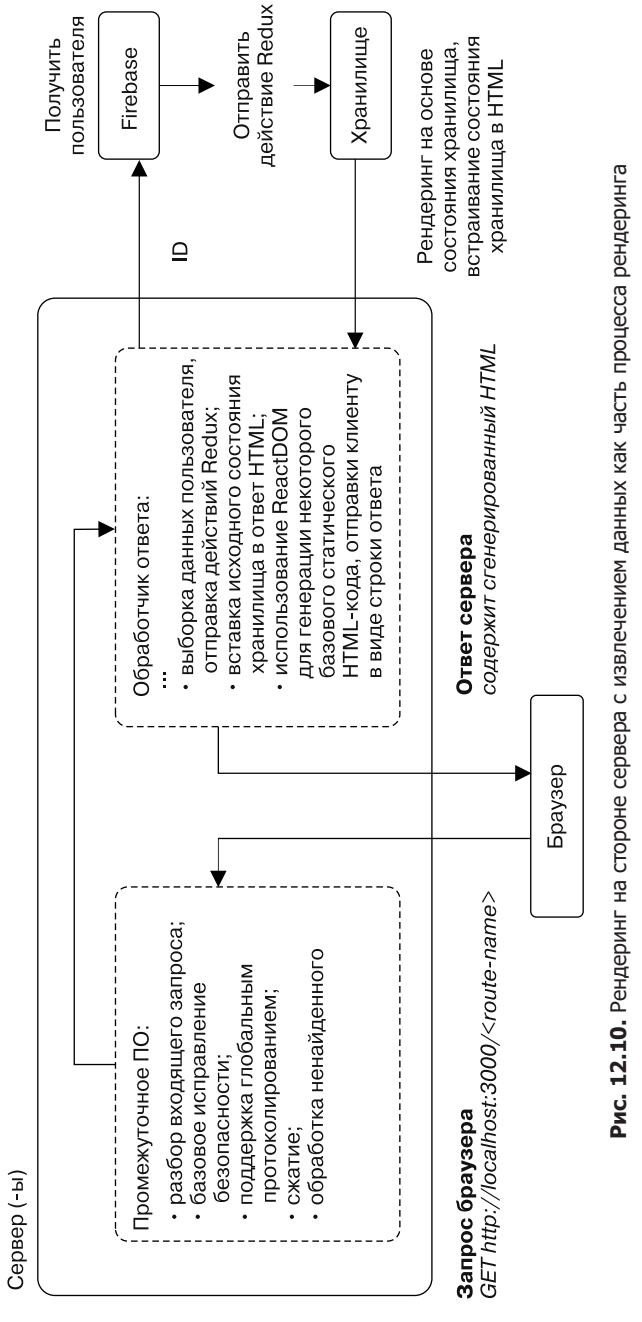


Рис. 12.10. Рендеринг на стороне сервера с извлечением данных как часть процесса рендеринга

Листинг 12.14. Получение данных для рендеринга на стороне сервера (server/server.js)

```
// ...
  const store = configureStore(initialReduxState);
  try {
    const token = req.cookies['letters-token'];
    if (token) {
      const firebaseUser = await firebase.auth()
        .verifyIdToken(token);
      const userResponse = await fetch(
        '${config.get('ENDPOINT')}/users/${firebaseUser.uid}'
      );
      if (userResponse.status !== 404) {
        const user = await userResponse.json();
        await store.dispatch(loginSuccess(user));
        await store.dispatch(getPostsForPage());.
      }
    }
  } catch (err) {
    if (err.errorInfo.code === 'auth/argument-error') {
      res.clearCookie('letters-token');
    }
    // Отправляем сообщение об ошибке
    store.dispatch(createError(err));
  }
  //...
```

Создание экземпляра хранилища Redux

Получение пользовательского токена из cookie

Проверка токена с помощью Firebase и применение ответа для извлечения пользователя из API JSON

Если пользователь существует, разворачивание ответа JSON из API (используются библиотека isomorphic-fetch и синтаксис async/await)

Если есть ошибка, например, истек токен, выполняется ее отправка в хранилище

Благодаря Redux-thunk допустима отправка создателей асинхронного действия, которые работают во время авторизации. Нужно дождаться их завершения, прежде чем переходить к следующему шагу

Это бóльшая часть работы, которую нужно сделать для рендеринга приложения с исключительно пользовательским контекстом! Один из недостатков этого подхода в том, что, если бы у вас было много страниц с разными требованиями к выборке данных, было бы сложно их согласовать. У вас нет способа сказать: «Ах, мы запрашиваем страницу *X*, а странице *X* нужны данные *Y*». Однако есть способы сделать это, и я кратко рассказываю о них в своем блоге, размещенном по адресу ifelse.io/2017/09/07/serverrendering-with-react-router-and-react-16-fiber (если вам интересно больше узнать об этом и о некоторых новых версиях React Router).

Чтобы завершить улучшения рендеринга, нужно сделать еще кое-что. Во-первых, требуется способ вставить строку HTML, которую вернет React-DOM. Поскольку он работает с потоками, подход с применением шаблона строки, который вы задействовали изначально, нужно изменить. Вместо непосредственной вставки полученного HTML-кода используйте для своего приложения две функции записи HTML. Одна из них будет содержать информацию заголовка, которая потребуется приложению (метаданные о приложении, данные Open Graph, ссылки CSS и т. д.). Другая — встраивать состояние хранилища Redux в ответ HTML. Вы хотите встроить состояние, чтобы при запуске браузер не переделывал работу, которую уже сделал сервер. Для этого нужно меньше рендеринга, а не больше! В листинге 12.15 показан компонент обертки HTML, в который вы передадите свой компонент и состояние хранилища Redux.

Листинг 12.15. Встраивание состояния Redux

```
Основные метаданные о приложении —
некий шаблонный код опущен
как не относящийся
к текущему обсуждению

 const ogProps = {
   updated_time: new Date(),
   type: 'website',
   url: 'https://social.react.sh',
   title: 'Letters Social | React in Action by
     Mark Thomas from Manning Publications',
   description:
     'Letters Social is a sample application for the React.js book React in
     Action by Mark Thomas from Manning Publications. Get it today at
     https://ifelse.io/book'
};

export const start = () => {

  return '<!DOCTYPE html><html lang="en-us">
    <head>
      <link rel="stylesheet" href="/static/styles.css" type="text/css" />
      <link rel="stylesheet" href="https://api.mapbox.com/
        mapbox.js/v3.1.1/mapbox.css" />
      <meta http-equiv="X-UA-Compatible" content="IE=edge" />
      <title>
        Letters Social | React in Action by Mark Thomas
        from Manning Publications
      </title>
      <link rel="manifest" href="/static/manifest.json" />
      <meta name="viewport" content="width=device-width,initial-scale=1" />
      <meta name="ROBOTS" content="INDEX, FOLLOW" />
      <meta property="og:title" content="${ogProps.title}" />
      <meta property="og:description" content="${ogProps.description}" />
      <meta property="og:type" content="${ogProps.type}" />
      <meta property="og:url" content="${ogProps.url}" />
      <meta property="og:updated_time" content="${ogProps.updated_time}" />
      <meta itemProp="description" content="${ogProps.description}" />
      <meta name="twitter:card" content="summary" />
      <meta name="twitter:title" content="${ogProps.title}" />
      <meta name="twitter:description" content="${ogProps.description}" />
      <meta property="book:author" content="Mark Tielens Thomas" />
      <meta property="book:tag" content="react" />
      <meta property="book:tag" content="reactjs" />
      <meta property="book:tag" content="React in Action" />
      <meta property="book:tag" content="javascript" />
      <meta property="book:tag" content="single page application" />
      <meta property="book:tag" content="Manning publications" />
      <meta property="book:tag" content="Mark Thomas" />
      <meta name="HandheldFriendly" content="True" />
      <meta name="MobileOptimized" content="320" />
      <meta name="theme-color" content="#4469af" />
```

Добавление приложения в основной контейнер div, поэтому, когда React-DOM берет на себя работу в браузере, ему не придется повторять работу сервера

```
        <link
          href="https://fonts.googleapis.com/
            css?family=Open+Sans:400,700,800" rel="stylesheet"
        />
      </head>
      <body>
        <div id="app">
    ';
};

export const end = reduxState => {
  return '</div>
    <script id="initialState">
      window.__INITIAL_STATE__ = ${JSON.stringify(reduxState)};
    </script>
    <script src="https://cdn.ravenjs.com/3.17.0/raven.min.js"
      type="text/javascript"></script>
    <script src="https://api.mapbox.com/mapbox.js/v3.1.1/mapbox.js"
      type="text/javascript"></script>
    <script src="/static/bundle.js" type="text/javascript"></script>
    </body>
  </html>';
};
```

Хранилище Redux в браузере должно быть в состоянии взять на себя работу, когда сервер остановлен, поэтому хранилище встраивается в формате JSON-stringified

При этом необходимо изменить хранилище Redux, чтобы оно могло взять работу на себя. В этом листинге вы сделаете следующее: убедитесь, что хранилище Redux каждый раз создается с нуля на сервере (чтобы предотвратить возможные ошибки, упомянутые ранее), и научите его читать начальное состояние из DOM. В листинге 12.16 перечислены незначительные изменения, которые вы внесете в свое рабочее хранилище (версия разработки не будет рендериться сервером, поэтому первоначальное состояние не будет найдено).

Листинг 12.16. Изменение хранилища Redux для SSR (src/store/configureStore.prod.js)

```
//...
let store;
export default function configureStore(initialState) {
  if (store && !isServer()) {
    return store;
  }
  const hydratedState =
    !isServer() && process.env.NODE_ENV === 'production'
      ? window.__INITIAL_STATE__
      : initialState;
  store = createStore(
    rootReducer,
    hydratedState,
    compose(applyMiddleware(thunk, crashReporting))
  );
  return store;
}
```

При нахождении на сервере каждый раз возвращается новое хранилище

При нахождении не на сервере с приложением в рабочем режиме состояние DOM проверяется и используется, если возможно

Теперь хранилище способно прочитать начальное состояние из данных, встроенных сервером, и не придется выполнять двойную работу. Что осталось? Как

говорилось в начале главы, у вас были асинхронные функции, доступные для рендеринга на стороне сервера. Сейчас работает метод `renderToString` из React-DOM, но он синхронный, и это может стать проблемой для сервера, если сразу многие воспользуются приложением. В React 16 был введен асинхронный вариант рендеринга на стороне сервера, который вы и примените. Процесс идентичен, за исключением потоков node.js, которые используются вместо синхронного метода.

Упражнение 12.2. Библиотеки с открытым исходным кодом

Вы интегрировали процессы рендеринга на стороне сервера в приложение Letters Social. У вас это работает с Redux, но масштабирование до очень большого приложения или введение новых требований к выборке данных (например, для других страниц) могут потребовать некоторой реорганизации и пересмотра подхода к рендерингу на стороне сервера. Существуют библиотеки с открытым исходным кодом для выполнения такого рендеринга с помощью React, они помогают устранять проблемы, связанные с универсальным рендерингом компонентов на стороне сервера. Чтобы разобраться в возможностях рендеринга на стороне сервера с помощью React, найдите время, чтобы изучить их и исходный код функций. Вы, вероятно, будете приятно удивлены тем, чего можете добиться с рендерингом на стороне сервера (оптимизированный рендеринг в случае react-server — github.com/redfin/react-server) и насколько абстракция способна упростить реализацию его принципов (в случае Next.js — github.com/zeit/next.js/).

Если вы раньше работали с node.js, то, вероятно, знакомы с потоками. Если нет, не переживайте. Потоки в node.js являются абстрактным интерфейсом для работы с потоковыми данными. Он может включать в себя такие процессы, как чтение или запись файла, преобразование и сжатие изображений или работа с HTTP-запросами и ответами. Подробнее о потоках в node.js узнайте по адресу nodejs.org/api/stream.html. Листинг 12.17 иллюстрирует использование нового API `renderToNodeStream` в React-DOM.

Листинг 12.17. Асинхронный рендеринг на стороне сервера (server/server.js)

```
res.setHeader('Content-type', 'text/html');
res.write(HTML.start());
const renderStream = renderToNodeStream(
  <Provider store={store}>
    <RouterContext {...props} />
  </Provider>
);
renderStream.pipe(res, { end: false });
renderStream.on('end', () => {
  res.write(HTML.end(store.getState()));
  res.end();
});
```

Запись заголовка Content-type, чтобы браузер знал, содержимого какого типа ожидать

Браузер должен начать загрузку страницы как можно быстрее, поэтому отправляется первая часть приложения

Создание потока приложения для рендеринга

Конвейер рендерит приложение для браузера, но пока не заканчивает поток

Когда поток выдает конечное событие и рендеринг выполнен, отправляется остальная часть HTML-кода и завершается ответ

Теперь Letters Social полностью рендерится для отображения пользователям. Вы можете наблюдать процесс, если используете инструменты разработчика для проверки загрузки документа и данных, отправляемых сервером (что-то подобное показано на рис. 12.11). Если запустите приложение в рабочем режиме, то заметите разницу в скорости, но анализ с помощью инструментов разработки в Chrome или Firefox позволит пошагово проверить загрузку приложения. Вы увидите, что сервер отправляет полную веб-страницу, а не просто страница рендерится после загрузки приложения.

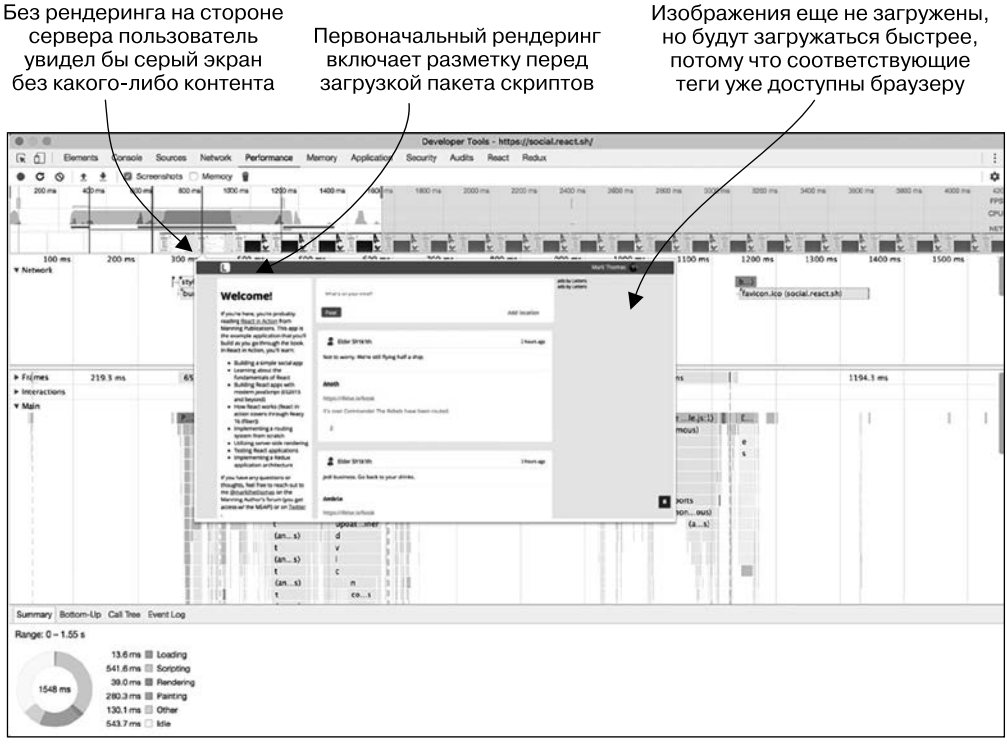


Рис. 12.11. Если открыть вкладку Performance (Производительность) инструментов разработчика Chrome на сайте social.react.sh, вы увидите, что сервер отправляет полностью визуализированный HTML-код и не дожидается загрузки пакета приложения для рендеринга приложения

12.8. Резюме

В этой главе мы рассмотрели, как реализовать рендеринг на стороне сервера в приложении. При этом может быть задействовано немало функций приложения, включая маршрутизацию, выборку данных и управление состоянием (Redux).

- Рендеринг на стороне сервера (SSR) генерирует статическую разметку для пользовательского интерфейса на сервере, которая отправляется клиенту. SSR с React

включает использование React-DOM для рендеринга строки HTML, которую React способна повторно применять в ходе работы на клиентской стороне, или статическую разметку (`ReactDOM.renderToString()`), которая должна оставаться статической в браузере (`ReactDOM.renderToStaticMarkUp()`).

- Не все фреймворки или библиотеки JS созданы для поддержки SSR. React является таковой и может захватить разметку, которая была сгенерирована на сервере, не требуя повторного первоначального рендеринга существующих элементов в браузере.
- Применение для маршрутизации решения, подобного React Router, позволит совместно использовать маршруты на клиенте и сервере, позволяя делиться кодом на разных платформах.
- SSR может быть сложным в реализации и годится только в определенных случаях. Некоторые ситуации, где это имеет смысл: если вы особенно обеспокоены SEO, у вас есть приложение, для которого критична быстрота первого отображения, или React задействуется как генератор статической разметки.
- Ожидаемого выигрыша в производительности от использования SSR часто можно достичь только в случае, если полезная нагрузка страницы, отправленная сервером, не слишком велика (чтобы загрузка не продолжалась дольше, чем прежде). Более длительное время отклика и больший объем данных могут снизить скорость первого отображения, которую вы получили бы в противном случае.
- SSR требует, чтобы вы рассмотрели, какие части приложения будут работать на сервере, а какие — нет. Функции, требующие среды браузера, должны быть исправлены для работы или обработаны так, чтобы не запускаться на сервере.
- Вы можете выполнить полный рендеринг на сервере, синхронизировав состояния аутентификации между клиентом и сервером и сделав необходимую выборку данных на сервере.
- Хотя существуют и другие реализации платформы JS, SSR требует, чтобы вы запустили сервер node.js или по крайней мере обратились к нему, чтобы сгенерировать HTML-код для отправки клиенту.

В следующей главе кратко рассмотрим React Native и завершим путешествие в изучение основ React.

13 Введение в React Native

- Обзор React Native.
- Различия между React и React Native.
- Источники информации о React Native.

На данный момент вы изучили основы React, реализовали роутер, освоили Redux, взглянули на рендеринг на стороне сервера и даже научились использовать библиотеку React Router. Что осталось? Есть еще много такого, чему стоит учиться и что исследовать в экосистеме React. В этой главе кратко рассматривается React Native — еще один проект в экосистеме React, разработанный Facebook. С помощью React Native вы можете писать React-приложения, которые запускаются на мобильных платформах, таких как iOS и Android. То есть разрабатывать приложения, которые запускаются на смартфонах и любых других платформах, на которые React Native ориентирован сейчас или будет ориентирован в дальнейшем. React Native дает разработчику отличный опыт создания мобильных React-приложений, и во многом поэтому он становится все более важным и популярным в сообществе React.

Поскольку React Native и начало работы над мобильной разработкой — это действительно большая область, я буду краток и сосредоточусь главным образом на общих концепциях. К концу главы вы будете иметь представление о том, что такое React Native и чем он хорош, и поймете, как изучить его глубже.

13.1. Обзор React Native

Прежде чем на сцене появился React Native, вы могли выбирать из нескольких вариантов, когда дело доходило до создания мобильных приложений. Можно было либо использовать платформы iOS и Android и известные вам языки, либо выбрать один из доступных гибридных подходов. Они различаются в реализации, но часто применяют веб-представление (читайте: мобильный браузер) и обеспечивают интерфейсы для формирования собственных SDK. Один из недостатков этого подхода таков: хотя вы можете писать собственные приложения, которые позволяют использовать многие знакомые веб-интерфейсы и идиомы, приложение не является действительно

нативным, и иногда наблюдается заметная разница в производительности и общем взаимодействии. Преимущество заключается в том, что даже команды или разработчики, не имеющие опыта разработки мобильных приложений, могли, задействуя навыки работы в Интернете, создавать мобильные приложения.

Тема мобильной разработки и того, как платформы, языки и аппаратные средства играют в ней разные роли, выходит за рамки данной книги. Но выбор между гибридными и общепринятыми подходами имеет отношение к обсуждению React Native, поскольку он предлагает новую альтернативу. С помощью React Native вы можете создавать действительно нативные приложения, а также использовать комбинацию JavaScript и кода для конкретной платформы (например, Swift или Java).

React Native стремится привнести идиомы и концепции построения пользовательских интерфейсов с помощью React в разработку мобильных приложений и задействовать наилучшие возможности мобильной разработки и разработки браузеров. Он поощряет совместное использование кода на разных платформах (есть компоненты, предназначенные как для iOS, так и для Android-устройств), позволяет писать нативный код там, где это целесообразно, и компилируется в нативное приложение — все это применяя множество схожих идиом, известных в React.

Давайте рассмотрим несколько особенностей React Native.

- С помощью React Native вы можете писать JavaScript-приложения, которые также могут использовать нативный код (Swift или Java), и компилировать их для нативных приложений, работающих под управлением операционной системы iOS или Android.
- React Native поддерживает создание одних и тех же элементов пользовательского интерфейса на Android и iOS, что потенциально упрощает разработку мобильных приложений.
- По желанию вы можете добавить собственный нативный код, поэтому не ограничены использованием только JavaScript.
- React-приложения Native задействуют идиомы React и следуют тем же декларативным компонентным концепциям, а в некоторых случаях даже API, при разработке пользовательского интерфейса.
- Инструментарий разработчика для выполнения React-приложений Native позволяет перезагрузить приложение с изменениями, не дожидаясь длинного цикла компиляции. Это часто экономит время разработчиков и упрощает работу.
- Возможность использовать разделяемый код и ориентироваться на несколько платформ иногда способна уменьшить количество разработчиков, необходимых для создания определенного приложения или проекта. Это может привести к сокращению количества кодовых баз, которые нуждаются в поддержке, а разработчики могут легко переходить между веб-сайтами и нативными платформами.
- В некоторых случаях вы можете делиться с программами React Native логикой и другими возможностями веб-приложений React, например бизнес-логикой и даже стилями.

Как работает React Native? Это способно показаться магией, черным ящиком — вот так взять JavaScript и получить скомпилированное нативное приложение. Вам не нужно знать, как работают отдельные составляющие React Native, чтобы использовать его функции, так же как не нужно знать все входы и выходы React-DOM для написания замечательных React-приложений. Но часто полезно хотя бы поверхностное понимание применяемой технологии.

С помощью React Native вы можете разрабатывать приложения, представляющие собой смесь JavaScript и нативного кода. React Native делает это реальным, создавая мост между вашим приложением и целевой мобильной платформой. Большинство мобильных устройств могут выполнять JavaScript, и React Native использует это преимущество для запуска вашего JavaScript-кода. Когда JavaScript выполняется вместе с каким-либо нативным кодом, система моста React Native задействует наряду с прочим базовую библиотеку React, чтобы перевести иерархию компонентов (с обработчиками событий, состоянием, свойствами и стилями) в представление на мобильном устройстве.

Когда происходит обновление (например, пользователь нажимает кнопку), React Native переводит нативное событие (касание, встряску, событие геолокации или что-то еще) в событие, которое может обрабатывать ваш JavaScript или нативный код. Он также отображает корректный пользовательский интерфейс на основе изменений в состоянии или свойствах. Вдобавок React Native помещает весь код в пакет и выполняет любую компиляцию, необходимую для того, чтобы вы могли опубликовать приложение в магазине Apple App Store или в Google Play Store.

Нюансов в работе этих процессов и в том, как функционирует React Native, намного больше, чем я упомянул, но основной процесс перевода между JavaScript, запущенным на устройстве, API-интерфейсами и событиями на платформе — это «волшебство» React Native. Результатом является платформа, с которой вы можете работать без снижения производительности. Это золотая середина между проблемами предыдущих гибридных подходов к мобильным приложениям, позволяющая избежать некоторых болезненных точек в традиционной мобильной разработке. На рис. 13.1 показан общий обзор процесса.

Похоже, что он отдаляется от React, который вы изучали в этой книге, и во многом это так и есть. Но сходство важнее различий. Я еще расскажу об этом в следующем разделе, но вы можете изучить код листинга 13.1, чтобы увидеть, насколько компонент React Native подобен компонентам, с которыми вы работали до сих пор.

В этой главе я не рассказываю, как настроить проект React Native и как работает код в листинге 13.1. Посетите страницу repl.it/KOAE/3, если хотите разобраться в этом и поэкспериментировать с React Native. Repl.it — это онлайн-платформа для запуска и совместного использования кода в интерактивном режиме, поддерживающая React Native. Вы сможете отсканировать QR-код смартфоном, чтобы просмотреть работу React-приложения Native. Это отличный способ поэкспериментировать с React Native, не выполняя настройку или конфигурирование.

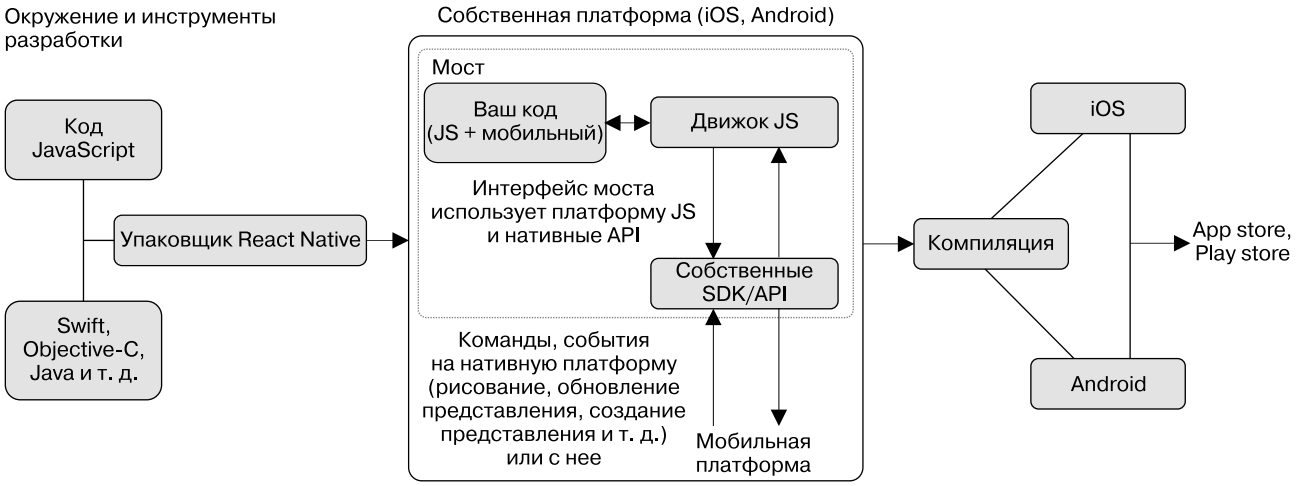


Рис. 13.1. React Native работает, создавая мост между вашим JavaScript и нативной базовой платформой. На большинстве таких платформ изначально реализованы виртуальная машина JavaScript или другой способ запуска JavaScript. Мост позволяет выполнять JavaScript-код вашего приложения. Система моста React Native будет ретранслировать сообщения между базовой платформой и JavaScript-кодом, чтобы нативные события могли быть переведены в поддерживаемые вашими компонентами React

Вы могли заметить кое-что важное — что элементы компонента (View, Text) аналогичны элементам `div` и `span` ваших компонентов из предыдущих разделов. Это пример общих концепций React, сохраняющихся на разных платформах. Не так уж важно, каковы отдельные элементы компонента, пока вы можете повторно их использовать и компоновать, как показано в листинге 13.1.

Листинг 13.1. Пример компонента React Native

```
import React, { Component } from 'react';
import { Text, View } from 'react-native';

export default class WhyReactNativeIsSoGreat extends Component {
  render() {
    return (
      <View>
        <Text>
          If you like React on the web, you'll like React Native.
        </Text>
        <Text>
          You just use native components like 'View' and 'Text',
          instead of web components like 'div' and 'span'.
        </Text>
      </View>
    );
  }
}
```

Вы все еще можете использовать обычный React.Component, даже в нативном приложении

React Native поставляется с базовыми составляющими для создания мобильных приложений

Text больше похож на контейнер span в браузере

Вы можете составлять компоненты с помощью React Native, компонент View выглядит как контейнер div в браузере (обычный компонент разметки)

Существуют и другие проекты, такие как React VR, где основной фокус еще дальше уходит от веб-интерфейсов, с которыми вы работали, но они используют те же шаблоны и концепции. Это одна из самых мощных возможностей платформы React, что особенно заметно, когда вы задействуете ее на разных платформах. Узнать больше о React VR можно на странице facebook.github.io/react-vr.

13.2. React и React Native

Насколько схожи React и React Native? Помимо того что их названия похожи, оба они используют базовую библиотеку React, но нацелены на разные платформы (браузеры и мобильные устройства). В этом разделе кратко рассмотрим некоторые их различия и сходство. Сравним важные характеристики React и React Native.

- *Среда выполнения.* React и React Native нацелены на разные платформы. React направлена на браузеры и, таким образом, использует API, специфичные для

браузера. Вы можете увидеть результаты этого в каждом API. Например, такие свойства, как класс, идентификатор и др., обычно встречаются в веб-компонентах React. Нативные платформы применяют разную семантику разметки и стилизации, поэтому вы не увидите многие из этих свойств в компонентах React Native. Также браузерные и мобильные приложения работают на разных типах устройств, поэтому не следует игнорировать такие явления, как потоки, загрузка процессора, и другие различия в базовой технологии, когда вы имеете дело с React и React Native.

- *Основные API.* Многие API, специфичные для React (например, используемые в жизненных циклах компонентов, состоянии, свойствах и т. д.), применимы и для React Native. Но каждая платформа реализует различные API для работы с сетью, разметкой, геолокацией, управлением ресурсами, сохранением, событиями и другими важными областями. React Native стремится импортировать некоторые знакомые API из мира, ориентированного на браузеры, например API Fetch для сетевого взаимодействия (developer.mozilla.org/ru/docs/Web/API/Fetch_API) и API Flexbox для компоновки (developer.mozilla.org/ru/docs/Web/CSS/flex). React Native также предоставляет события, но они более специфичны для мобильных платформ (например, `onPress`). Это может стать незначительным препятствием, но, к счастью, есть библиотеки, которые помогают устранить различия между веб-интерфейсами и нативными API, такие как `react-primitives` (github.com/lelandrichardson/react-primitives).
- *Компоненты.* В веб-проекте React нет встроенных компонентов (например, для изображений, разметки текста или других элементов пользовательского интерфейса). Вы создаете их сами. А React Native включает компоненты для текста, представлений, изображений и т. д. Это примитивы, которые нужны для написания пользовательских интерфейсов для мобильных приложений и аналогичны элементам DOM для среды браузера.
- *Использование базовой библиотеки React.* Как React, так и React Native работают с базовой библиотекой React для определения компонентов. В каждом из проектов применяется своя система рендеринга для соединения всего вместе и взаимодействия с устройством — браузером или мобильным. Для Всемирной паутины React использует библиотеку `React-Dom`, а React Native реализует собственную систему. Такой подход позволяет вам писать компоненты одинаковыми способами для разных платформ.
- *Методы жизненного цикла.* Компоненты React Native также имеют методы жизненного цикла, поскольку наследуются от одного и того же базового класса `React`, и эти методы поддерживаются конкретной системой платформы — React-DOM или React Native.
- *Типы событий.* В то время как React-DOM реализует синтетическую систему событий, которая позволяет компонентам работать с событиями браузера стандартным способом, мобильные приложения используют другие события. Одним из

примеров является жест. На сенсорных устройствах вы можете панорамировать, масштабировать, перетаскивать и выполнять многие другие жесты. Компоненты, написанные в компонентах React Native, позволяют вам реагировать на эти события.

- *Стилизация оформления.* Поскольку React Native не нацелен на браузеры, вам нужно немного иначе настроить свои компоненты. При обычной мобильной разработке нет API-интерфейса CSS, но вы можете применить большинство свойств CSS с помощью React Native. Он предоставляет специальный API, где полное соответствие между свойствами невозможно. Возьмем, например, анимацию CSS. Спецификация CSS и способы реализации браузеров отличаются от того, как iOS и Android поддерживают и реализуют анимацию, поэтому вам нужно анимировать иначе и использовать правильный API для каждой платформы. Изучение новых API для стилизации, возможно, займет много времени и помешает прямому совместному применению стилей CSS в веб-проектах и нативных проектах. К счастью, есть библиотеки, которые работают с React и React Native, например `styled-components` (www.styled-components.com). С ростом популярности React Native будет разработано больше таких кроссплатформенных библиотек.
- *Сторонние зависимости.* Как и в случае с React, для React Native можно использовать сторонние библиотеки компонентов. Как отмечалось ранее, многие популярные библиотеки, такие как `React Router` и `styled-components`, содержат варианты, нацеленные на React Native. Одна из наиболее привлекательных возможностей React Native заключается в том, что он все еще способен задействовать модульную экосистему JavaScript.
- *Распространение.* Вы можете развернуть React-приложения практически в любом современном браузере, однако React-приложения Native требуют специфических для платформы инструментов распространения как для разработки, так и для окончательной версии (например, Xcode). Обычно следует использовать процесс сборки React Native для компиляции приложения при окончательной загрузке. «Огороженный сад», характерный для инструментов iOS и Android, — это хорошо известный компромисс для разработки мобильных приложений.
- *Инструментарий разработки.* React для Всемирной паутины действует в браузерах, поэтому можете применять любые инструменты для работы с браузером, которые помогут при разработке и отладке. Для React Native не обязательно иметь специальную инструментальную платформу, но она может оказаться полезной. Одно из ключевых различий между проектами заключается в том, что React Native фокусируется на горячей перезагрузке, которая по умолчанию не является частью React. *Горячая перезагрузка* способна ускорить разработку мобильных устройств, потому что не нужно ждать, пока приложение будет компилироваться. На рис. 13.2 показаны некоторые инструменты разработчика, к которым вы получаете доступ, работая с React Native.

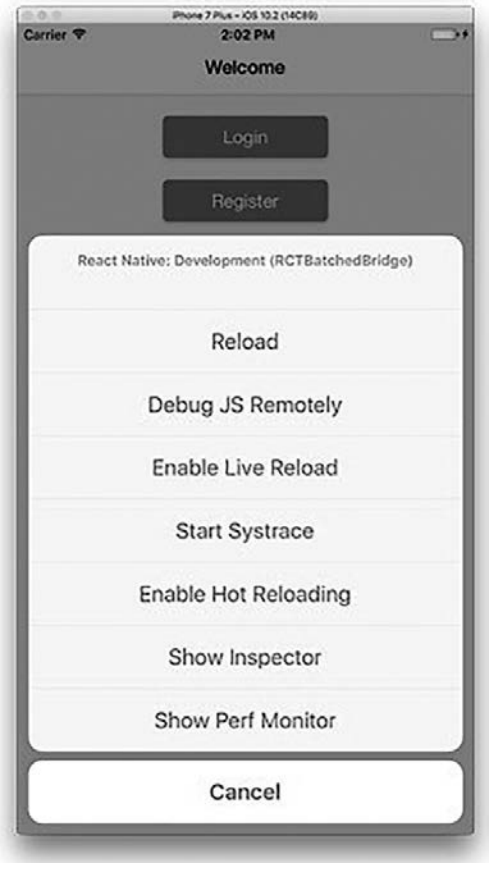


Рис. 13.2. React Native поставляется с рядом дополнительных инструментов разработчика, которые помогают в повышении производительности, отладке и реализации другой функциональности. Их наличие означает также, что у вас меньше ограничений на применение таких инструментов для разработки, как Xcode, хотя вы, безусловно, можете использовать свои инструменты для разработки на специфической платформе. Есть много причин, по которым React Native как технология был воспринят особенно хорошо, и одна из них — получение разработчиками богатого опыта

13.3. Когда использовать React Native

Не каждый разработчик и не каждая команда нуждаются в React Native. Представим несколько сценариев, в реализации которых вы могли бы участвовать, и посмотрим, окажется при этом React Native полезным или нет.

- *Один разработчик.* Если вы только начали изучать React или задействуете ее для побочных проектов, то, вероятно, знакомитесь с React Native для удовольствия или работы в каких-то мобильных проектах. Также React Native следует рассмотреть,

если вы недостаточно глубоко разбираетесь в нативной разработке и хотите упростить ее или создать более простое приложение. Если вы уже знаете React, имеет смысл погрузиться в использование React Native для мобильной разработки с некоторыми знакомыми концепциями, которыми вы уже пользовались.

- *Небольшая кросс-функциональная команда.* В маленьких стартапах разработчики часто трудятся над широким спектром задач, начиная с серверных приложений и заканчивая клиентскими (веб-приложения, мобильные приложения и т. д.). В подобных ситуациях React Native может помочь разработчикам, занятым множеством дел, добиться того, чтобы организация трудилась над мобильным приложением, не имея серьезного опыта работы с мобильными устройствами, и продолжала использовать свои наработки в React. Этот прием можно применять и в крупных организациях, в которых хотят легко перемещать разработчиков между приложениями или проектами.
- *Команда с небольшим или средним опытом в нативных разработках.* Если вы или ваша команда плохо знакомы с принципами мобильной разработки, но изучили React и JavaScript, React Native поможет просто и быстро реализовать продукт. Не нужно подменять существующий опыт, не нужно переходить на Swift (iOS) или Java (Android), это может сэкономить ваше время.
- *Значительный опыт работы с нативными приложениями.* Некоторые команды выбирают React Native не потому, что в какой-то мере он понижает планку для разработки мобильных технологий, а потому, что помогает стандартизировать идиомы и шаблоны в различных реализациях приложений для бизнеса (для мобильных и настольных компьютеров). Но если это не проблема, у вас уже есть значительный опыт и на мобильную разработку затрачено много времени, может потребоваться более тщательная оценка, чтобы понять, выиграет ли ваша команда от применения доступных абстракций и шаблонов.

Помимо соображений о команде и опыте, которые нужно учитывать, думая о React Native, вы также должны принимать во внимание некоторые ограничения, присущие технологии в том виде, в каком она существует сегодня.

- *Использование JavaScript.* Если в вашей команде или организации нет разработчиков на JavaScript или уже имеется большой опыт разработки мобильных приложений, может быть, и нет смысла переводить разработчиков на JavaScript и его экосистему, и это нормально. Как и React для Всемирной паутины, React Native не является «серебряной пулей» и должен оцениваться на основе компромиссов, а не шумихи вокруг него.
- *Специфические потребности в производительности.* React Native производителен, но в качестве абстракции может стать еще одним препятствием для достижения определенной производительности, к которой стремитесь вы или ваша команда. Например, если визуализация 3D-сцен является основной целью приложения, React Native, вероятно, будет не лучшим решением. Другие фреймворки, например Unity, подходят лучше. Это согласуется с тем, что React не является «серебряной пулей», о чем я только что упомянул и что пытался внушить в предыдущих главах.

- *Узкоспециализированное приложение*. Некоторые типы приложений не подходят для модели React. Приложения с расширенной реальностью (AR), графические или другие узкоспециализированные приложения часто требуют особых библиотек и навыков, которыми не обладают большинство веб-разработчиков. Это не означает, что их невозможно создать, но в данный момент React Native не фокусируется на удовлетворении таких потребностей.
- *Внутреннее применение*. Иногда крупные компании разрабатывают приложения для внутреннего использования, которые помогают сотрудникам лучше выполнять свою работу. React Native может пригодиться для таких приложений, потому что они обычно имеют относительно простой интерфейс и могут быть быстро доработаны программистами, не специализирующимися на разработке мобильных устройств.

Конечно, вам и вашей команде решать, стоит ли применять технологию в конкретном случае, но, надеюсь, теперь вы лучше представляете, когда есть смысл использовать React Native.

13.4. Простейший пример Hello World

Я не буду рассказывать о том, как интегрировать React Native в Letters Social, однако в этом разделе уделю некоторое внимание базовому примеру Hello World, чтобы можно было увидеть его в действии. Вы будете работать за пределами репозитория Letters Social, поэтому размещайте код приложения там, где удобно. Для начала выполните команды, показанные в листинге 13.2.

Листинг 13.2. Установка приложения create-react-native-app

```
cd ./path-to-your-react-native-sample-folder

npm install -g create-react-native-app

create-react-native-app
```

Запустив эти команды, вы увидите несколько файлов, созданных в соответствующей папке, и некоторые инструкции. Эти команды аналогичны доступным в приложении Create React App — таком же проекте, только ориентированном на React.js для веб-платформы. Вы можете больше узнать о приложении Create React App на странице github.com/facebookincubator/create-react-app. На рис. 13.3 показано, что вы должны увидеть, начав работать с библиотекой Create React Native App.

Инструмент Create React Native App установил зависимости, выполнил файлы шаблонов, настроил процесс сборки и интегрировал в проект инструментарий Expo React Native. SDK Expo наряду с прочим расширяет функциональность React Native и упрощает работу с аппаратными технологиями. Среда разработки Expo XDE позволяет легко управлять несколькими проектами React Native, а также создавать и развертывать их.

Рис. 13.3. Когда приложение запускается в режиме разработки, вы должны увидеть запуск упаковщика React Native и сообщение, подобное показанному здесь. Следуйте инструкциям, чтобы убедиться, что на вашем локальном компьютере установлен Expo XDE. В зависимости от того, на какую среду вы ориентируетесь, откройте эмулятор либо операционной системы Android, либо iOS

Вы не напишете ничего сверхъестественного, но можете попробовать понять, как легко разрабатывать приложения с помощью React Native. После того как упаковщик React Native будет запущен с помощью команды `yarn start`, откройте один из эмуляторов — Android или iOS, чтобы видеть запущенное приложение. Поменяйте

часть кода шаблона и посмотрите, как работает горячая перезагрузка. В листинге 13.3 показан простой компонент, который извлекает некоторые данные из API Star Wars при монтировании. Обратите внимание на то, что React Native уже использует современные веб-API, такие как Flexbox и Fetch.

Листинг 13.3. Простой пример React Native (App.js)

```
import React from 'react';
import { StyleSheet, Text, View } from 'react-native';

export default class App extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      people: []
    };
  }
  async componentDidMount() {
    const res = await fetch('https://swapi.co/api/people');
    const { results } = await res.json();
    this.setState(() => {
      return {
        people: results
      };
    });
  }
  render() {
    return (
      <View style={styles.container}>
        <Text style={{ color: '#fcd433', fontSize: 40, padding: 10 }}>
          A long time ago, in a Galaxy far, far away...
        </Text>
        <Text>Here are some cool people:</Text>
        {this.state.people.map(p => {
          return (
            <Text style={{ color: '#fcd433' }} key={p.name}>
              {p.name}
            </Text>
          );
        })}
      </View>
    );
  }
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
    backgroundColor: '#000',
    alignItems: 'center',
    justifyContent: 'center'
  }
});
```

В отличие от React, React Native поставляется с примитивными компонентами для пользовательского интерфейса

Конструктор, инициализация состояния и методы жизненного цикла одинаковы в React и React Native

Вы также можете использовать современные функции JavaScript, такие как async/await, в React-приложениях Native

Несмотря на то что в React Native стили похожи, вы не применяете CSS

Выражения JSX одинаковы в React Native и React

Создание таблицы стилей в React Native требует использования своего API Stylesheet для стилизации компонентов

Если вы внесете изменения в приложение, то увидите, как упаковщик обновляет его в реальном времени (рис. 13.4). Надеюсь, это даст вам представление о том, как легко создавать приложения в React Native. Вы можете задействовать горячую перезагрузку для веб-приложений, но для мобильной разработки цикл компиляции – проверки — перекомпиляции способен занять значительное время.

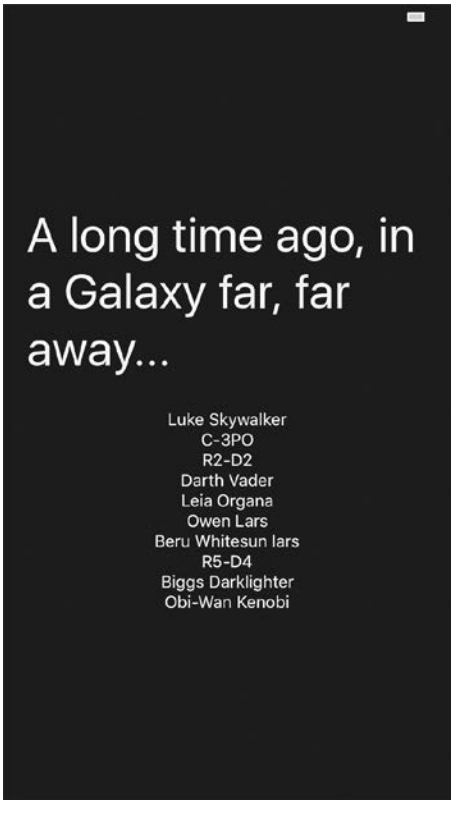


Рис. 13.4. Изменения мгновенно отражаются в эмуляторе, в котором запущен код приложения

Итак, вы создали свой первый компонент React Native и код, который дает краткое представление о том, как работает технология и как легко это происходит.

13.5. Дальнейшее изучение

Одна из фраз, которую вы увидите в документации React и на сайте сообщества: *«Учите один раз, пишите где угодно»*. Она напоминает фразу: *«Напишите один раз, запускайте где угодно»*, популярную в сообществе Java, и это одна из отличительных черт парадигмы React. Как я сказал ранее, вы можете изучить концепции React и применять их на различных платформах — от Сети до мобильных и VR. По мере

ознакомления с тем, как использовать React на новой платформе, вы увидите отличия и нюансы, специфичные для платформы, но бóльшую часть ваших знаний о React можно будет применить без проблем. Это одна из причин того, почему работа с React способна оказаться очень приятной.

Существует много источников информации, к которым вы можете обратиться, если хотите продолжить изучение React Native. Один из них — книга *React Native in Action* Надера Дабита (Nader Dabit) (Manning Publications, 2018), которая хорошо согласуется с данной книгой, потому что это отличное введение в React Native (рис. 13.5). Применив свои знания, полученные из этой книги, вы сможете создавать мобильные приложения с помощью React Native. К тому же она станет хорошим источником знаний, если ваша команда рассматривает применение React Native в предстоящем проекте.

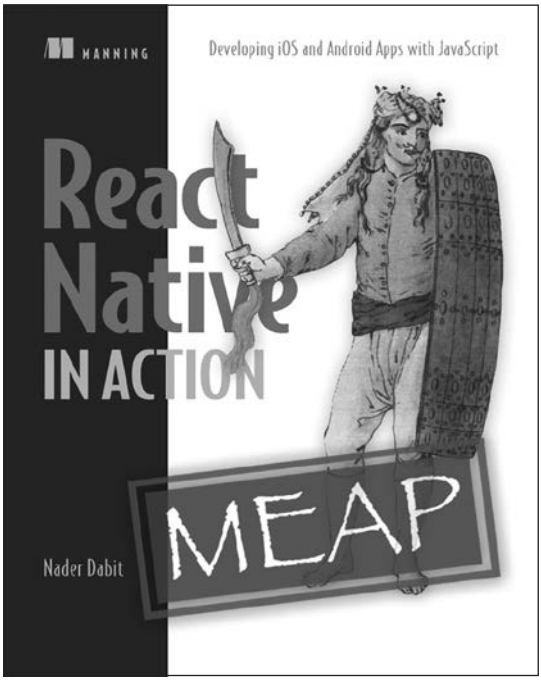


Рис. 13.5. Книга Надера Дабита дает навыки по iOS, Android и веб-разработке, необходимые для создания надежных сложных приложений React Native. Если вы продолжаете интересоваться React, эта книга идеальна для того, чтобы прочесть ее следующей. Подробнее см. на сайте www.manning.com/books/react-native-in-action

Еще один отличный ресурс, который поможет вам начать работу с React Native, — проект Create React Native App. Это отличное место для запуска нового проекта React Native и шикарный пример приложения для новичков. Проект включает в себя несколько предустановленных библиотек и инструменты для создания приложений React Native, но позволяет выгружать и восстанавливать значения по умолчанию.

Если вы интересуетесь приложениями Create React App или Create React Native App, посетите следующие ресурсы в Интернете:

- *Create React Native App* — github.com/react-community/create-reactnative-app;
- *Create React App* — github.com/facebook/create-react-app;
- *документация React Native* — facebook.github.io/react-native.

13.6. Резюме

Вот краткое описание того, что вы узнали из этой главы.

- React Native — это технология в экосистеме React, которую разработчики могут использовать для написания React-приложений, работающих на мобильных устройствах iOS и Android.
- React Native применяет базовую библиотеку React для создания компонентов, но другой набор библиотек — для поддержки рендеринга приложения на нативной платформе и взаимодействия с базовой платформой (события касания, геолокация, доступ к камерам и т. д.).
- React Native управляет мостом между JavaScript-кодом разработчика и базовой мобильной платформой.
- React Native использует много API, которые идентичны или похожи на веб-API. Он применяет Flexbox для разметки, Fetch для сетевых запросов и другие знакомые API.
- Вы можете смешивать JavaScript и нативный код при создании приложений React Native.
- React Native предоставляет надежный набор инструментов для разработки и компиляции приложений.
- Инструменты разработчика React Native для быстрой перезагрузки экономят время, позволяя не ждать, пока приложение будет перекомпилироваться.
- Применение React Native снижает планку для разработки мобильных устройств для вас или вашей команды.
- Вы не будете использовать React Native для абсолютно всех типов мобильных приложений, но для большинства типичных мобильных приложений его должно быть достаточно.
- Книга *React Native in Action* Надера Дабита (Manning Publications, 2018) — отличный ресурс, который можно рассмотреть в своем дальнейшем путешествии по React. Ознакомьтесь с ней на сайте www.manning.com/books/react-native-in-action.

Марк Тиленс Томас

React в действии

Перевел с английского С. Черников

Заведующая редакцией	*Ю. Сергиенко*
Руководитель проекта	*О. Сивченко*
Ведущий редактор	*Н. Гринчик*
Литературный редактор	*Н. Рощина*
Художественный редактор	*С. Заматевская*
Корректор	*Е. Павлович*
Верстка	*Г. Блинов*

Изготовлено в России. Изготовитель: ООО «Прогресс книга».
Место нахождения и фактический адрес: 194044, Россия, г. Санкт-Петербург,
Б. Сампсониевский пр., д. 29А, пом. 52. Тел.: +78127037373.

Дата изготовления: 11.2018. 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 07.11.18. Формат 70×100/16. Бумага офсетная. Усл. п. л. 29,670. Тираж 1200. Заказ 0000.

Отпечатано в ОАО «Первая Образцовая типография». Филиал «Чеховский Печатный Двор».
142300, Московская область, г. Чехов, ул. Полиграфистов, 1.
Сайт: www.chpk.ru. E-mail: marketing@chpk.ru
Факс: 8(496) 726-54-10, телефон: (495) 988-63-87

ИЗДАТЕЛЬСКИЙ ДОМ ПИТЕР®
WWW.PITER.COM

ВАША УНИКАЛЬНАЯ КНИГА

Хотите издать свою книгу? Она станет идеальным подарком для партнеров и друзей, отличным инструментом для продвижения вашего бренда, презентом для памятных событий! Мы сможем осуществить ваши любые, даже самые смелые и сложные, идеи и проекты.

МЫ ПРЕДЛАГАЕМ:

- издать вашу книгу
- издание книги для использования в маркетинговых активностях
- книги как корпоративные подарки
- рекламу в книгах
- издание корпоративной библиотеки

Почему надо выбрать именно нас:

Издательству «Питер» более 20 лет. Наш опыт — гарантия высокого качества.

Мы предлагаем:

- услуги по обработке и доработке вашего текста
- современный дизайн от профессионалов
- высокий уровень полиграфического исполнения
- продажу вашей книги во всех книжных магазинах страны

Обеспечим продвижение вашей книги:

- рекламой в профильных СМИ и местах продаж
- рецензиями в ведущих книжных изданиях
- интернет-поддержкой рекламной кампании

Мы имеем собственную сеть дистрибуции по всей России, а также на Украине и в Беларуси. Сотрудничаем с крупнейшими книжными магазинами. Издательство «Питер» является постоянным участником многих конференций и семинаров, которые предоставляют широкую возможность реализации книг.

Мы обязательно проследим, чтобы ваша книга постоянно имелась в наличии в магазинах и была выложена на самых видных местах.

Обеспечим индивидуальный подход к каждому клиенту, эксклюзивный дизайн, любой тираж.

Кроме того, предлагаем вам выпустить электронную книгу. Мы разместим ее в крупнейших интернет-магазинах. Книга будет сверстана в формате ePub или PDF — самых популярных и надежных форматах на сегодняшний день.

Свяжитесь с нами прямо сейчас:

Санкт-Петербург – *Анна Титова, (812) 703-73-73, titova@piter.com*
Москва – *Сергей Клебанов, (495) 234-38-15, klebanov@piter.com*