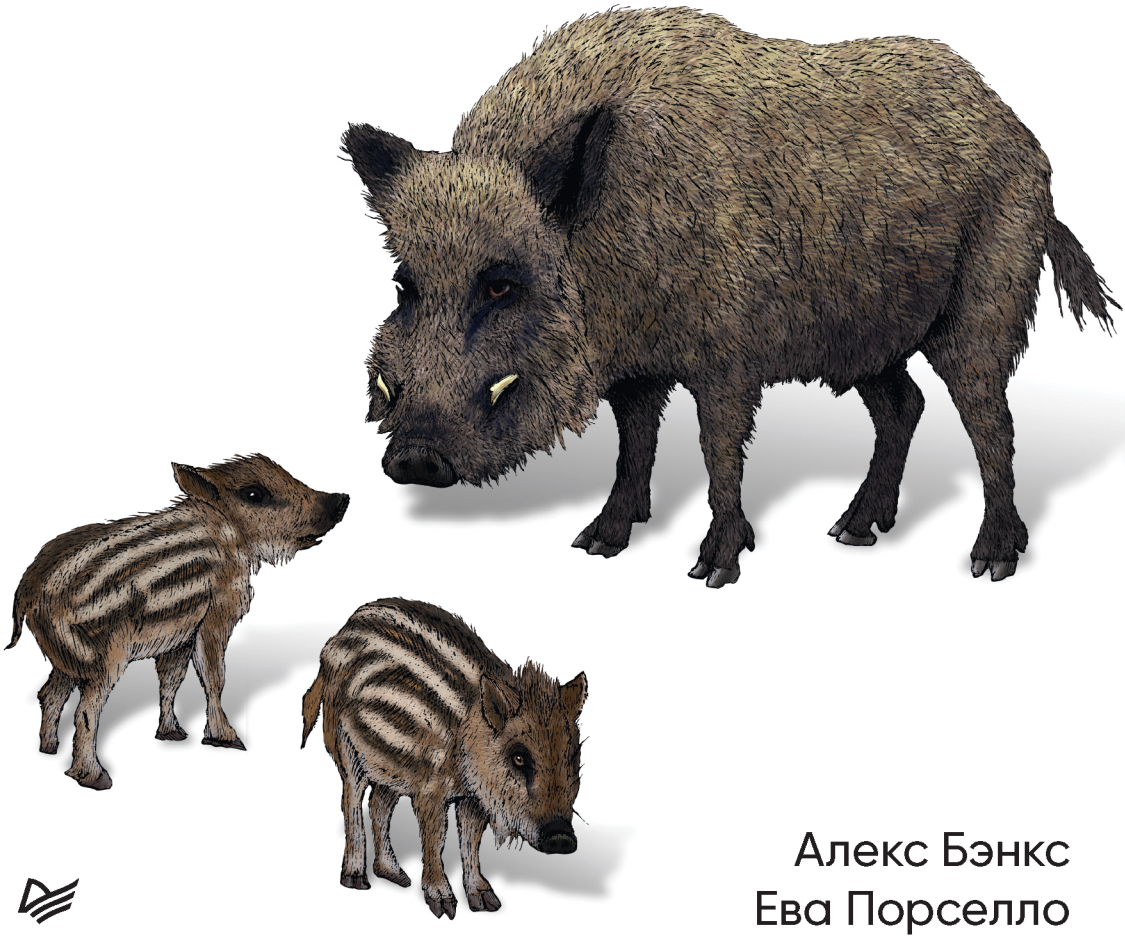
O'REILLY®

Второе издание

React

Современные шаблоны для разработки приложений



Алекс Бэнкс
Ева Порселло

SECOND EDITION

Learning React

Modern Patterns for Developing React Apps

Alex Banks and Eve Porcello

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

React

Современные шаблоны для разработки приложений

2-е издание

Алекс Бэнкс
Ева Порселло

ПИТЕР®

Санкт-Петербург • Москва • Минск

2022

ББК 32.988.02-018
УДК 004.738.5
Б97

Бэнкс Алекс, Порселло Ева

Б97 React: современные шаблоны для разработки приложений. 2-е изд. — СПб.: Питер, 2022. — 320 с.: ил. — (Серия «Бестселлеры O'Reilly»).

ISBN 978-5-4461-1492-4

Хотите создавать эффективные приложения с помощью React? Тогда эта книга написана для вас. Познакомьтесь лучшими практиками и шаблонами создания современного кода.

Вам не потребуются глубокие знания React или функционала JavaScript — достаточно знакомства с принципами работы JavaScript, CSS и HTML.

Алекс Бэнкс и Ева Порселло научат вас создавать пользовательские интерфейсы, которые будут динамически отображать изменения без необходимости перезагрузки страницы даже на крупномасштабных сайтах, работающих с огромными массивами данных.

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

ББК 32.988.02-018
УДК 004.738.5

ISBN 978-1492051725 англ.

Authorized Russian translation of the English edition of Learning React 2E ISBN 9781492051725 © 2020 Alex Banks and Eve Porcello.
This translation is published and sold by permission of O'Reilly Media, Inc., which owns or controls all rights to publish and sell the same.

ISBN 978-5-4461-1492-4

Краткое содержание

Оглавление

Предисловие

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

Книга не предполагает наличия у читателей начальных знаний React. Мы познакомим вас со всеми основами React с нуля. Также мы не предполагаем, что вы работали с новейшим синтаксисом JavaScript. О нем мы поговорим в главе 2, заложив тем самым фундамент для остальных глав.

Освоение книги станет для вас легче, если вы уже знакомы с HTML, CSS и JavaScript. Почти всегда лучше освоиться с этой большой тройкой, прежде чем погружаться в работу с библиотекой JavaScript.

В ходе изучения материала можете обращаться к репозиторию GitHub по адресу github.com/moonhighway/learning-react. Там находится весь код, позволяющий освоить практические примеры.

Условные обозначения

В этой книге приняты следующие типографские соглашения:

Курсив

Используется для обозначения новых терминов, адресов URL и электронной почты, имен файлов и расширений имен файлов.

`Моноширинный шрифт`

Применяется для оформления листингов программ и программных элементов внутри обычного текста, таких как имена переменных и функций, баз

данных, типов данных, переменных окружения, инструкций и ключевых слов.

`Моноширинный жирный`

Обозначает команды или другой текст, который должен вводиться пользователем.

`Моноширинный курсив`

Обозначает текст, который должен замещаться фактическими значениями, вводимыми пользователем или определяемыми из контекста.

Так обозначаются советы или предложения.

Так обозначаются примечания общего характера.

Так обозначаются предупреждения.

Использование примеров кода

Дополнительные материалы (примеры кода, упражнения и т. д.) можно скачать по ссылке github.com/moonhighway/learning-react.

Если у вас возникнет технический вопрос или проблема с использованием примеров кода, отправьте электронное письмо по адресу bookquestions@oreilly.com.

Эта книга призвана помочь вам выполнить свою работу. Обычно, если в этой книге предлагается пример кода, вы можете использовать его в своих программах и документации. Вам не нужно связываться с нами для получения разрешения, если вы не воспроизводите слишком значительную часть кода. Например, для написания программы, в которой используется несколько фрагментов кода из этой книги, разрешения не требуется. Продажа или распространение примеров из книг O'Reilly требует разрешения. Чтобы ответить на вопрос, можно сослаться на эту книгу и процитировать пример кода без разрешения. Для включения

значительного количества примеров кода из этой книги в документацию по вашему продукту требуется разрешение.

Благодарности

Наше путешествие по React началось случайно. Когда в стенах компании Yahoo мы преподавали полный курс по разработке программ на JavaScript, то использовали для создания учебных материалов библиотеку YUI. А затем в августе 2014 года разработка на YUI прекратилась. Пришлось заменить все файлы нашего курса, но чем? Что теперь предполагалось использовать при разработке внешнего интерфейса? Ответ: React. В него мы влюбились не сразу, понадобилась пара часов на то, чтобы он зацепила нас. Казалось, React потенциально может изменить что угодно. Мы включились в работу с ним на ранней стадии и считаем, что нам очень повезло.

Мы благодарим Анджелу Руфино и Дженнифер Поллок за поддержку в разработке второго издания. Также хотим поблагодарить Элли Макдональд за всю ее помощь в редактировании первого издания. Мы благодарны нашим научным редакторам Скотту Ивако, Адаму Ракису, Брайану Слеттену, Максу Фиртману и Четану Каранде.

Кроме того, эта книга не состоялась бы без Шэрон Адамс и Мэрилин Мессинео. Они сговорились приобрести первый компьютер Алекса — цветной Tandy TRS 80. Также эта книга не возникла бы без любви, поддержки и поощрения Джима и Лорри Порселло, Майка и Шэрон Адамс.

Также хочется поблагодарить кофейню *Coffee Connexion* из Тахо-Сити, штат Калифорния, за их кофе, такой нужный для завершения этой книги, и владельца кофейни Робина, который дал нам не теряющий актуальности комментарий: «Книга по программированию? Скучища!»

От издательства

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

Мы будем рады узнать ваше мнение!

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

ГЛАВА 1

Добро пожаловать в React

Что делает библиотеку JavaScript хорошей? Количество звезд на GitHub? Количество загрузок на npm? Насколько важно количество твитов, которые оставляют ребята из ThoughtLeaders? Как выбрать лучший инструмент для создания лучшего продукта? Как узнать, стоит ли овчинка выделки? Как сделать выбор?

Когда появилась библиотека React, было много разговоров о том, хороша ли она, и многие относились к ней скептически. Она была новой, а новое часто разочаровывает.

В ответ на такую реакцию Пит Хант из команды React написал статью «Почему React?», в которой порекомендовал «дать ей [библиотеке React] пять минут». Он хотел побудить людей сначала поработать с React, а потом оценивать его.

Да, React — это небольшая библиотека, в которой нет всего, что вам может понадобиться для создания приложения. Но дайте ей пять минут.

Да, код на React выглядит как HTML, расположенный прямо в коде JavaScript. И да, эти теги требуют предварительной обработки для запуска в браузере. И для этого вам, вероятно, понадобится такой инструмент сборки, как webpack. Но дайте ей пять минут.

Скоро React отметит десятилетие, и многие команды уже решили, что она хороша, так как в свое время дали ей пять минут. Мы говорим об Uber, Twitter и Airbnb — огромных компаниях, которые попробовали React и поняли, что он может помочь командам быстрее создавать лучшие продукты. В конце концов, разве не для этого мы все здесь? Не для твитов. Не для звезд. Не для загрузок. Мы здесь, чтобы создавать крутые вещи с помощью инструментов, которые нам нравятся. Мы здесь ради славы от создания чего-то, чем мы можем гордиться. Если вам нравится делать такие вещи, вам, вероятно, понравится и React.

Прочный фундамент

Независимо от того, новичок вы в React или ищете в этой книге новейшие функции, мы хотим, чтобы эта книга стала прочной основой для всей вашей будущей работы с этой библиотекой. Цель книги — предотвратить путаницу и расположить материал в определенной последовательности, то есть предоставить дорожную карту обучения React.

Прежде чем углубиться в React, важно знать JavaScript. То есть освоить не весь язык и все его паттерны, но обязательно понимать массивы, объекты и функции.

В следующей главе мы рассмотрим новый синтаксис JavaScript, чтобы познакомить вас с новейшими функциями JavaScript, особенно с теми, которые часто используются с React. Затем мы познакомим вас с функциональным JavaScript, чтобы вы смогли понять парадигму, которая породила React. Хорошим побочным эффектом работы с React является то, что он может сделать вас более сильным разработчиком JavaScript, так как с его помощью вы начнете создавать паттерны, которые удобно читать, повторно использовать и тестировать. Он вас немного воспитает.

Затем мы рассмотрим основы React, чтобы понять, как создать пользовательский интерфейс с помощью компонентов. Научимся составлять эти компоненты и добавлять к ним логику с помощью свойств и состояний. Рассмотрим хуки React, позволяющие повторно использовать логику с отслеживанием состояния между компонентами.

Разобравшись с основами, мы создадим приложение, в котором пользователи смогут добавлять, редактировать и удалять цвета. Мы узнаем, как Hooks и Suspense помогают получить данные. В процессе создания приложения мы изучим множество инструментов из более широкой экосистемы React, которые используются для решения общих задач, таких как маршрутизация, тестирование и рендеринг (или отображение, глава 7) на стороне сервера.

Мы надеемся, что вы быстрее освоите экосистему React с использованием нашего подхода, так как мы не только проведем поверхностный обзор тем, но и вооружим вас инструментами и навыками, необходимыми для создания реальных приложений React.

Прошлое и будущее React

Библиотека React была создана Джорданом Уолком, инженером-программистом Facebook. Она попала в новостную ленту Facebook в 2011 году, а затем в Instagram, приобретенный Facebook в 2012 году. На JSConf 2013 React была

представлена как система с открытым исходным кодом и присоединилась к обширной категории UI-библиотек, таких как jQuery, Angular, Dojo, Meteor и другие. В то время React называли «V в MVC». Другими словами, компоненты React выступали в качестве уровня представления или пользовательского интерфейса для приложений JavaScript.

С этого момента React начали принимать в сообществе. В январе 2015 года компания Netflix объявила, что использует React для разработки пользовательского интерфейса. В том же месяце была выпущена React Native — библиотека для создания мобильных приложений с использованием React. Facebook также выпустил ReactVR — еще один инструмент, решавший с помощью React широкий круг задач рендеринга. В 2015 и 2016 годах появилось огромное количество популярных инструментов, таких как React Router, Redux и Mobx, для обработки маршрутизации и управления состоянием. В конце концов, React стала позиционироваться как библиотека, поскольку она была связана с реализацией определенного набора функций, а не являлась инструментом для решения отдельных задач.

Еще одной серьезной вехой в развитии React стал выпуск React Fiber в 2017 году. Fiber — это волшебным образом переписанный алгоритм рендеринга React. Полная переработка части React, которая практически ничего не изменила в общедоступном API, позволила сделать React более современной и производительной, не затрагивая пользователей.

Совсем недавно, в 2019 году, вышел Hooks — новый способ добавления и совместного использования логики с отслеживанием состояния между компонентами. Мы также застали выпуск Suspense — способа оптимизации асинхронного рендеринга с помощью React.

В будущем мы обязательно увидим еще больше изменений, но одной из причин успеха React является сильная команда, которая работала над проектом на протяжении многих лет. Эта команда амбициозна, но осторожна, продвигает дальновидные оптимизации, постоянно учитывая влияние любых изменений на библиотеку, которые будут каскадно распространяться по сообществу.

В React и связанные с ним инструменты могут быть внесены критические изменения. Поэтому будущие версии этих инструментов могут нарушить работоспособность некоторых примеров кода в этой книге. Но вы все равно сможете ознакомиться с этими примерами. Мы предоставим точную информацию о версии в файле package.json, чтобы вы могли установить пакеты.

Также вы можете следить за изменениями в официальном блоге React по адресу facebook.github.io/react/blog. О выпуске новых версий React чтайте в блоге основной команды ее разработчиков и журналах изменений. Блог переводится на все

большее число языков, локальные версии документов находятся на странице сайта документации по адресу reactjs.org/languages.

Изменения во втором издании книги

Это *второе издание* книги. Мы сочли важным обновить книгу, потому что React значительно изменился за последние несколько лет. Мы сосредоточимся на передовых методах, которые отстаивает команда React, но упомянем и устаревшие функции. Код, написанный на React много лет назад с использованием старых стилей, все еще хорошо работает и должен поддерживаться. Мы будем отмечать устаревшие функции на боковой панели.

Работа с файлами

В этом разделе мы обсудим, как работать с файлами для этой книги и как установить некоторые полезные инструменты React.

Файловый репозиторий

В репозитории GitHub для этой книги (github.com/moonhighway/learning-react) приведены все файлы кода, систематизированные по главам.

React Developer Tools

Мы настоятельно рекомендуем установить React Developer Tools для работы с React. Эти инструменты доступны как расширение браузера для Chrome и Firefox и как отдельное приложение для использования с Safari, IE и React Native. После установки инструментов разработчика вы сможете просматривать дерево компонентов React, их свойства, сведения об их состоянии и даже сайты, используемые React в настоящее время в производственной среде. Это по-настоящему полезно при отладке и изучении проектов на основе React.

Чтобы установить эти инструменты, перейдите на репозиторий GitHub (oreil.ly/5tizT). Там вы найдете ссылки на расширения для Chrome (oreil.ly/Or3pH) и Firefox (oreil.ly/uw3uv).

После установки React Developer Tools вы увидите, какие сайты используют React, с помощью загорающегося значка на панели инструментов браузера (рис. 1.1).

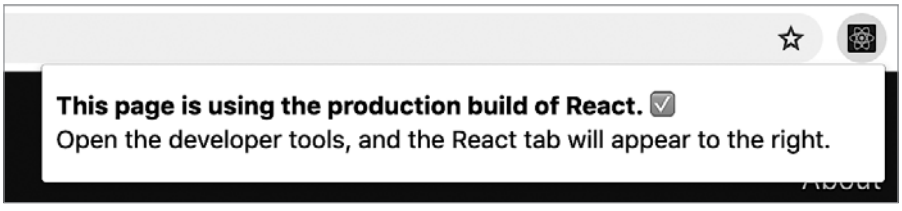


Рис. 1.1. Просмотр React Developer Tools в Chrome

Когда вы откроете React Developer Tools, появится новая вкладка под названием React (рис. 1.2). Нажав на нее, вы увидите компоненты, составляющие просматриваемую страницу.

Рис. 1.2. Проверка DOM с помощью React Developer Tools

Установка Node.js

Node.js — это среда выполнения JavaScript, используемая для создания полнофункциональных приложений. У Node открытый исходный код, и ее можно установить на Windows, macOS, Linux и другие платформы. Мы будем использовать Node в главе 12 при сборке сервера.

У вас должен быть установлен Node, но вам не обязательно быть экспертом по Node, чтобы использовать React. Если вы не уверены, что на вашем компьютере установлен Node.js, откройте окно терминала или командной строки и введите команду:

```
node -v
```

После запуска этой команды вы должны увидеть номер версии Node, желательно 8.6.2 или выше. Если вы увидите сообщение об ошибке `Command not found`, значит, Node.js не установлен. Это легко исправить, установив Node.js с сайта проекта (nodejs.org). Просто выполните указания по установке, и, когда вы снова введете команду `node -v`, то увидите номер версии.

npm

Вместе с Node.js автоматически установится npm — менеджер пакетов Node. В сообществе JavaScript инженеры совместно используют проекты с открытым исходным кодом, чтобы избежать необходимости самостоятельно переписывать фреймворки, библиотеки или вспомогательные функции. Сам React — это пример прикладной библиотеки npm. В этой книге мы будем использовать npm для установки множества пакетов.

Большинство проектов JavaScript, которые можно встретить сегодня, содержат множество файлов и, в числе прочего, файл package.json. В нем описан проект и его зависимости. Если вы запустите команду `npm install` в папке, содержащей файл package.json, npm установит все пакеты, перечисленные в проекте.

Если вы начинаете собственный проект с нуля и хотите включить зависимости, просто выполните команду:

```
npm init -y
```

Эта команда инициализирует проект и создаст файл package.json. После этого вы сможете установить свои зависимости с помощью npm. Чтобы установить пакет с помощью npm, запустите команду:

```
npm install имя-пакета
```

Чтобы удалить пакет с помощью npm, запустите команду:

```
npm remove имя-пакета
```

Yarn

Альтернативой npm является Yarn. Этот менеджер пакетов был выпущен в 2016 году Facebook в сотрудничестве с Exponent, Google и Tilde. Он помогает Facebook и другим компаниям надежно управлять своими зависимостями. Если вы знакомы с рабочим процессом npm, разобраться с Yarn не составит для вас труда. Сначала установите Yarn глобально с помощью npm:

```
npm install -g yarn
```

После этого вы будете готовы к установке пакетов. При установке зависимостей из pack.json вместо команды `npm install` вы можете запустить команду `run yarn`.

Чтобы установить конкретный пакет с помощью Yarn, запустите команду:

```
yarn add имя-пакета
```

Для удаления зависимости используйте соответствующую команду:

```
yarn remove имя-пакета
```

Yarn используется в рабочем процессе Facebook и входит в такие проекты, как React, React Native и Create React App. В проекте, использующем Yarn, есть файл yarn.lock. Чтобы установить все зависимости такого проекта, вместо `npm install` наберите `yarn`.

Теперь, когда ваша среда для разработки React настроена, вы готовы приступить к изучению React. В главе 2 мы познакомимся с новейшим синтаксисом JavaScript, который чаще всего встречается в коде React.

ГЛАВА 2

JavaScript для React

С момента своего появления в 1995 году JavaScript претерпел множество изменений. Поначалу он использовался для интерактивных элементов веб-страницы: нажатия кнопок, проверки наведения, проверки формы и т. д. Позже JavaScript стал надежнее благодаря появлению технологий DHTML и AJAX. Сегодня, когда есть Node.js, он стал настоящим языком программирования, который используется для создания полнофункциональных приложений. JavaScript везде.

Эволюцией JavaScript руководили представители компаний, использующих его, поставщики браузеров и лидеры сообщества. Комитет, отвечающий за изменения в JavaScript на протяжении многих лет, — Европейская ассоциация производителей компьютеров (ECMA). Изменения в языке инициируются сообществом и исходят из предложений, написанных членами сообщества. Кто угодно может направить свое предложение в комитет ECMA (https://tc39.es/process-document/). Обязанность комитета ECMA — заниматься этими предложениями: определять их приоритеты и решать, какие из них будут включены в спецификацию.

Первый выпуск ECMAScript произошел в 1997 году под названием ECMAScript1. За ним в 1998 году последовал ECMAScript2. ECMAScript3 вышел в 1999 году, и в нем появились регулярные выражения, обработка строк и многое другое. Процесс согласования ECMAScript4 превратился в хаотичный политический беспорядок, и выпуск так и не состоялся. В 2009 году вышел ECMAScript5 (ES5), в котором были такие функции, как новые методы массива, свойства объектов и поддержка библиотеки для JSON.

С тех пор согласования происходят активнее. После выпуска ES6 или ES2015 в 2015 году новые функции JavaScript выпускаются ежегодно. Все, что является частью стадии предложения, обычно называется ESNext — другими словами, следующий элемент, который будет частью спецификации JavaScript.

Предложения проходят через четко определенные этапы: от этапа 0, на котором находятся новейшие предложения, до этапа 4, на котором представлены гото-

вые функции. Когда предложение начинает проходить эти этапы, поставщики браузеров, такие как Chrome и Firefox, должны реализовать соответствующие функции. Например, при появлении ключевого слова `const` (подробнее об этом позже в этой главе), поставщики браузеров внесли соответствующие изменения для его поддержки, чтобы иметь возможность использовать его в коде JavaScript.

Многие функции, которые мы обсудим в этой главе, уже поддерживаются новейшими браузерами, но мы также расскажем, как транспилировать код JavaScript, то есть преобразовать новый синтаксис в более старый синтаксис, который понимает браузер. В таблице совместимости (oreil.ly/oe7la) можно найти много информации о последних функциях JavaScript и степени их поддержки браузерами.

В этой главе мы покажем весь синтаксис JavaScript, который будем использовать в книге. Эти базовые знания синтаксиса JavaScript помогут вам в работе с React. Если вы еще не перешли на последнюю версию синтаксиса, сейчас самое время сделать это. Если вы уже знакомы с новейшими языковыми функциями, переходите к следующей главе.

Объявление переменных

До появления ES2015 объявить переменную можно было только с помощью ключевого слова `var`. Теперь у нас есть несколько вариантов объявления, обеспечивающих более широкие функциональные возможности.

Ключевое слово const

Константа — это переменная, которую нельзя переопределить. После ее объявления вы не можете изменить ее значение. Такие переменные часто используются в JavaScript. Как и в других языках, в JavaScript константы появились с выходом ES6.

До начала использования констант все переменные можно было переопределять:

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

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

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

Рис. 2.1. Попытка переопределить константу

Ключевое слово let

В современном JavaScript есть *область видимости лексических переменных*. Мы создаем блоки кода с помощью фигурных скобок ({}). В функциях эти фигурные скобки закрывают область видимости любой переменной, объявленной с помощью `var`. В то же время вспомните инструкции `if…else`. Если до настоящего момента вы работали на других языках программирования, то можете предположить, что в этих блоках блокируется и область видимости переменных. Но это не так.

Если переменная создается внутри блока `if…else`, она не будет привязана к блоку:

```
var topic = "JavaScript";

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

console.log("global", topic); // global React
```

Переменная `topic` внутри блока `if` переопределяет значение `topic` вне блока.

При наличии ключевого слова `let` мы можем ограничить переменную любым блоком кода. Использование `let` защищает значение глобальной переменной:

```
var topic = "JavaScript";

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

console.log("global", topic); // JavaScript
```

Значение переменной `topic` не переопределяется вне блока.

Еще одна область, где фигурные скобки не блокируют область видимости переменной, — циклы `for`:

```
var div,
  container = document.getElementById("container");

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

В этом цикле мы создаем пять блоков `div`, которые появятся внутри общего контейнера. Каждому `div` назначается обработчик события щелчка `onclick`, который создает окно с отображением индекса. Объявление переменной `i` в цикле `for` создает глобальную переменную с именем `i`, а затем выполняет итерацию по ней, пока ее значение не достигнет `5`. Когда вы щелкаете по любому из этих полей, появляется сообщение, что `i` равно `5` для всех `div`, потому что текущее значение глобального `i` равно `5` (рис. 2.2).

Рис. 2.2. Переменная i равна 5 в каждом поле

Объявление счетчика цикла `i` с помощью `let` вместо `var` блокирует область видимости переменной `i`. Теперь щелчок по любому окну отобразит значение для `i`, которое было привязано к итерации цикла (рис. 2.3):

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

Область видимости переменной `i` защищена оператором `let`.

Рис. 2.3. Ограничение области действия переменной i с помощью оператора let

Шаблонные строки

Шаблонная строка — это альтернатива конкатенации строк. Она позволяет вставлять переменные в строку. Также ее называют шаблоном литерала или строковым шаблоном.

Традиционная конкатенация строк — это использование знака + для создания строки со значениями переменных и строк:

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

С помощью шаблона мы можем создать одну строку и вставить значения переменных, заключив их в ${ }:

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

Любой код JavaScript, возвращающий значение, может быть добавлен в шаблонную строку между ${ и }.

Шаблонные строки учитывают пробелы, что упрощает создание шаблонов электронной почты, примеров кода или чего-либо еще, что содержит пробелы. Таким образом вы можете использовать шаблонную строку, занимающую несколько строк, не разбивая код:

```
const email = `
Hello ${firstName},

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

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

You can pick your tickets up 30 minutes before the show.
```

```
Thanks,

${ticketAgent}
`
```

Раньше, чтобы использовать HTML-строку непосредственно в коде JavaScript, нужно было прописывать ее целиком в одной строке. Теперь, когда пробелы распознаются как текст, вы можете вставить отформатированный код HTML, легкий для чтения и понимания:

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

Обратите внимание, что в этот код можно добавить переменные для заголовка страницы и текста статьи.

Создание функций

Чтобы с помощью JavaScript выполнить задачу повторно, используйте функцию. Рассмотрим некоторые варианты синтаксиса, которые можно использовать для создания функции, и содержание функций.

Объявление функций

Объявление (или определение) функции начинается с ключевого слова `function`, за которым следует имя функции: `logCompliment`. Операторы JavaScript, являющиеся частью функции, определяются между фигурными скобками:

```
function logCompliment() {
  console.log("You're doing great!");
}
```

Когда вы объявили функцию, ее можно вызвать, чтобы увидеть, как она выполняется:

```
function logCompliment() {
  console.log("You're doing great!");
}

logCompliment();
```

После вызова этой функции вы увидите в консоли комплимент.

Функциональные выражения

Вместо объявления функции можно использовать функциональное выражение. В этом случае функция создается как переменная:

```
const logCompliment = function() {
  console.log("You're doing great!");
};

logCompliment();
```

Результат тот же — в консоли появится комплимент.

При выборе между объявлением функции и функциональным выражением помните, что первые записываются в память, а вторые — нет. Другими словами, вы можете вызвать функцию до того, как объявите ее, но не можете вызвать функцию, созданную с помощью функционального выражения, так как это вызовет ошибку. Например:

```
// Invoking the function before it’s declared
hey();
// Function Declaration
function hey() {
  alert("hey!");
}
```

Такой код сработает. Вы увидите, что в браузере появится предупреждение. Здесь функция поднимается и перемещается в верхнюю часть области видимости файла. Выполнение того же кода с функциональным выражением вызовет ошибку:

```
// Invoking the function before it's declared
hey();
// Function Expression
const hey = function() {
  alert("hey!");
};
TypeError: hey is not a function
```

Это примитивный пример, но ошибка `TypeError` может иногда возникать при импорте файлов и функций в проект. Если она появилась, вы всегда можете выполнить рефакторинг кода и использовать объявление.

Передача аргументов

Сейчас функция `logCompliment` не принимает аргументы или параметры. Чтобы предоставить функции динамические переменные, мы можем передать именованные параметры функции, просто добавив их в круглых скобках. Начнем с добавления первой переменной `Name`:

```
const logCompliment = function(firstName) {
  console.log(`You're doing great, ${firstName}`);
};

logCompliment("Molly");
```

Теперь, когда мы вызовем функцию, значение переменной `firstName` будет добавлено в консоль.

Можно продолжить улучшать код, создав дополнительный аргумент с именем `message`. Теперь мы не будем жестко кодировать сообщение, а передадим динамическое значение в качестве параметра:

```
const logCompliment = function(firstName, message) {
  console.log(`${firstName}: ${message}`);
};

logCompliment("Molly", "You're so cool");
```

Возвращаемое значение функции

Наша функция `logCompliment` пишет комплимент в консоль, но чаще мы будем использовать функцию для возврата значения. Добавим к этой функции оператор `return`. Он будет указывать значение, возвращаемое функцией. Переименуем функцию в `createCompliment`:

```
const createCompliment = function(firstName, message) {
  return `${firstName}: ${message}`;
};

createCompliment("Molly", "You’re so cool");
```

Чтобы проверить, правильно ли работает функция, оберните ее вызов в `console.log`:

```
console.log(createCompliment("You're so cool", "Molly"));
```

Параметры по умолчанию

Многие языки, включая C++ и Python, позволяют разработчикам задавать значения по умолчанию для аргументов функций. Значения по умолчанию

появились в спецификации ES6, согласно которой в случае, если значение для аргумента не указано, используется значение по умолчанию.

Например, можно настроить строки по умолчанию для параметров `name` и `activity`:

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

Если функции `logActivity` не будут переданы аргументы, она сработает как надо, используя значения по умолчанию. Значения аргументов по умолчанию могут быть любого типа, не обязательно строками:

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

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

Стрелочные функции

Стрелочные функции — это еще одно полезное нововведение ES6. С помощью стрелочных функций вы можете создавать функции без использования ключевого слова `function`. Вам также часто не нужно использовать ключевое слово `return`. Рассмотрим функцию, которая принимает аргумент `firstName` и возвращает строку, превращая любого человека в лорда:

```
const lordify = function(firstName) {
  return `${firstName} of Canterbury`;
};

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

С помощью стрелочной функции мы можем значительно упростить синтаксис:

```
const lordify = firstName => `${firstName} of Canterbury`;
```

Эта стрелка создает полноценное объявление функции в одной строке. Ключевое слово `function` не используется. Мы также убрали слово `return`, потому что

стрелка указывает на то, что нужно вернуть. Причем, если функция принимает только один аргумент, можно убрать скобки вокруг аргументов.

При использовании более одного аргумента нужны круглые скобки:

```
// Typical function
const lordify = function(firstName, land) {
  return `${firstName} of ${land}`;
};
// Arrow Function
const lordify = (firstName, land) => `${firstName} of ${land}`;

console.log(lordify("Don", "Piscataway")); // Don of Piscataway
console.log(lordify("Todd", "Schenectady")); // Todd of Schenectady
```

Функцию удалось выразить в одной строке, поскольку возвращению подлежал результат только одного выражения. Если тело функции состоит более чем из одной строки, то его следует заключить в фигурные скобки:

```
const lordify = (firstName, land) => {
  if (!firstName) {
    throw new Error("A firstName is required to lordify");
  }

  if (!land) {
    throw new Error("A lord must have a land");
  }

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

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

Инструкции `if...else` заключены в квадратные скобки, но синтаксис стрелочной функции полезен и для них тоже.

Возврат объектов

Что произойдет, если вы захотите вернуть объект? Рассмотрим функцию с именем `person`, которая создает объект на основе параметров, переданных в аргументы `firstName` и `lastName`:

```
const person = (firstName, lastName) =>
    {
        first: firstName,
        last: lastName
    }

console.log(person("Brad", "Janson"));
```

Как только вы запустите этот код, увидите ошибку: `Uncaught SyntaxError: Unexpected token:`. Чтобы исправить ее, просто заключите возвращаемый объект в круглые скобки:

```
const person = (firstName, lastName) => ({
  first: firstName,
  last: lastName
});

console.log(person("Flad", "Hanson"));
```

Недостающие круглые скобки породили бесчисленное множество ошибок в приложениях JavaScript и React, поэтому не забывайте о них!

Стрелочные функции и область видимости

Обычные функции не блокируют оператор `this`. Например, `this` становится чем-то другим в обратном вызове `setTimeout`, а не объектом `tahoe`:

```
const tahoe = {
  mountains: ["Freel", "Rose", "Tallac", "Rubicon", "Silver"],
  print: function(delay = 1000) {
    setTimeout(function() {
      console.log(this.mountains.join(", "));
    }, delay);
  }
};

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

Эта ошибка возникает из-за того, что мы пытаемся использовать метод `.join` для объекта `this`. Но журнал говорит, что `this` является объектом `Window`:

```
console.log(this); // Window {}
```

Чтобы решить эту проблему, мы можем использовать синтаксис стрелочной функции для защиты области видимости оператора `this`:

```
const tahoe = {
  mountains: ["Freel", "Rose", "Tallac", "Rubicon", "Silver"],
  print: function(delay = 1000) {
    setTimeout(() => {
      console.log(this.mountains.join(", "));
    }, delay);
  }
};

tahoe.print(); // Freel, Rose, Tallac, Rubicon, Silver
```

Все работает правильно, и мы можем использовать для курортов метод `.join`, перечисляя их через запятую. Будьте осторожны, всегда помните об области видимости. Стрелочные функции не блокируют область видимости оператора `this`:

```
const tahoe = {
  mountains: ["Freel", "Rose", "Tallac", "Rubicon", "Silver"],
  print: (delay = 1000) => {
    setTimeout(() => {
      console.log(this.mountains.join(", "));
    }, delay);
  }
};

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

Замена функции `print` на стрелочную функцию означает, что `this` на самом деле является объектом `Window`.

Транспиляция JavaScript

Когда новая функция JavaScript предлагается и получает поддержку, сообщество часто стремится использовать ее до того, как она начнет поддерживаться всеми браузерами. Единственный способ убедиться, что код с новой функцией будет работать, — это преобразовать его перед запуском в браузере в более совместимый код. Этот процесс называется *транспиляцией*. Одним из самых популярных инструментов для транспиляции JavaScript является Babel (www.babeljs.io).

В прошлом использовать новейшие функции JavaScript можно было только подождав пару недель, месяцев или даже лет, пока браузеры не начнут поддерживать их. Сегодня Babel позволяет сразу же использовать новейшие функции JavaScript. Транспиляция делает JavaScript похожим на другие языки. Но нас интересует не совсем традиционная транспиляция, так как наш код приобретает не двоичный формат, а синтаксис, который интерпретируется большим числом браузеров. Кроме того, у JavaScript теперь есть исходный код, а это означает, что некоторые файлы в вашем проекте не будут запускаться в браузере.

В качестве примера рассмотрим стрелочную функцию с несколькими аргументами по умолчанию:

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

Если мы передадим этот код транспилятору Babel, он сгенерирует следующее:

```
"use strict";
var add = function add() {
  var x =
```

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

Babel добавил объявление `"use strict"` для работы в строгом режиме. Переменные `x` и `y` устанавливаются по умолчанию с использованием массива аргументов — техники, с которой вы, возможно, знакомы. Полученный код JavaScript лучше поддерживается браузерами.

Чтобы больше узнать о Babel, познакомьтесь с Babel REPL на сайте документации (babeljs.io/repl). Вы можете ввести новый синтаксис слева, а справа появится преобразованный код в старом формате.

Процесс транспиляции JavaScript обычно автоматизируется с помощью инструмента сборки, вроде web-pack или Parcel. Позже мы обсудим их подробнее.

Объекты и массивы

Начиная с ES2016, синтаксис JavaScript поддерживает весьма творческие способы определения переменных в объектах и массивах. Эти способы широко используются в сообществе React. Рассмотрим некоторые из них, в том числе деструктуризацию, расширение объектного литерала и оператор распространения.

Деструктуризация объектов

Деструктурирующее присваивание позволяет локально определять область видимости полей внутри объекта и объявлять, какие значения будут использоваться. Рассмотрим объект `sandwich`. У него четыре ключа, но мы хотим использовать значения только двух из них. Допустим, `bread` и `meat` будут доступны локально:

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

const { bread, meat } = sandwich;

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

Код извлекает из объекта `bread` и `meat` и создает для них локальные переменные. Кроме того, поскольку мы объявили эти переменные с помощью `let`, их можно изменить без изменения исходного сэндвича:

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

let { bread, meat } = sandwich;

bread = "garlic";
meat = "turkey";

console.log(bread); // garlic
console.log(meat); // turkey

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

Попробуем деструктурировать входные аргументы функции. Рассмотрим нашу функцию, которая называет пользователя лордом:

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

const regularPerson = {
  firstname: "Bill",
  lastname: "Wilson"
};

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

Вместо того чтобы использовать синтаксис точечной нотации для изучения объектов, мы деструктурируем нужные значения из объекта `regularPerson`:

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

const regularPerson = {
  firstname: "Bill",
  lastname: "Wilson"
};

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

Продвинемся на один уровень дальше и проследим за изменениями данных. Теперь у объекта `regularPerson` есть новый вложенный объект на ключе `spouse`:

```
const regularPerson = {
  firstname: "Bill",
```

```
  lastname: "Wilson",
  spouse: {
  firstname: "Phil",
    lastname: "Wilson"
  }
};
```

Чтобы присвоить звание лорда ему тоже, нужно слегка скорректировать деструктурированные аргументы функции:

```
const lordify = ({ spouse: { firstname } }) => {
  console.log(`${firstname} of Canterbury`);
};

lordify(regularPerson); // Phil of Canterbury
```

Используя двоеточие и вложенные фигурные скобки, мы можем деструктурировать поле `firstname` из объекта `spouse`.

Деструктуризация массивов

Также можно деструктурировать значения из массивов. Представьте, что вы хотите присвоить первое значение массива имени переменной:

```
const [firstAnimal] = ["Horse", "Mouse", "Cat"];

console.log(firstAnimal); // Horse
```

Можно передать ненужные значения через *сопоставление со списком* с помощью запятых, обозначающих элементы, которые следует пропускать. Имея тот же массив, мы можем получить доступ к последнему значению, заменив первые два значения запятыми:

```
const [, , thirdAnimal] = ["Horse", "Mouse", "Cat"];

console.log(thirdAnimal); // Cat
```

Позже в этом разделе мы объединим на этом примере деструктуризацию массива и оператор распространения.

Расширение объектного литерала

Расширение объектного литерала противоположно деструктуризации. Это процесс реструктуризации или воссоздания объекта. Он позволяет брать переменные из глобальной области видимости и добавлять их к объекту:

```
const name = "Tallac";
const elevation = 9738;

const funHike = { name, elevation };

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

Объекты `name` и `elevation` теперь являются ключами объекта `funHike`.

С помощью расширения объектного литерала или реструктуризации также можно создавать методы объекта:

```
const name = "Tallac";
const elevation = 9738;
const print = function() {
  console.log(`Mt. ${this.name} is ${this.elevation} feet tall`);
};

const funHike = { name, elevation, print };

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

Обратите внимание, что мы используем оператор `this` для доступа к ключам объекта.

При определении методов объекта больше не нужно использовать ключевое слово `function`:

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

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

```
    console.log("speed:", mph);
  }
};
```

Расширение объектного литерала позволяет включать глобальные переменные в объекты и сокращает количество текста, делая ключевое слово `function` ненужным.

Оператор распространения

Оператор распространения (`...`) выполняет несколько задач. Во-первых, он позволяет комбинировать содержимое массивов. Например, с его помощью можно создать массив, в котором объединены два других массива:

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

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

Все предметы из массивов `peaks` и `canyons` попали в новый массив с названием `tahoe`.

Рассмотрим, как оператор распространения решает другие задачи. Используя массив `peaks` из предыдущего примера, представим, что мы хотим получить последний элемент из массива, а не первый. Можно использовать метод `Array.reverse`, изменяющий порядок элементов массива на обратный, в сочетании с деструктуризацией массива:

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

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

Видите, что произошло? Функция `reverse` фактически изменила массив. Но когда у нас есть оператор распространения, нам не нужно изменять исходный массив. Вместо этого мы можем создать копию массива и обратить ее:

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

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

Поскольку для копирования массива мы использовали оператор распространения, массив `peaks` остался нетронутым и может быть использован позже в исходной форме.

Оператор распространения также можно использовать для получения оставшихся элементов в массиве:

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

const [first, ...others] = lakes;

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

Также можно использовать синтаксис с тремя точками для сбора аргументов функции в массив. Внутри функции они будут считаться *прочими параметрами*. Создадим функцию, которая принимает *n* аргументов с помощью оператора распространения, а затем использует эти аргументы для печати некоторых консольных сообщений:

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

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

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

Функция `directions` принимает аргументы с помощью оператора распространения. Первый аргумент присваивается переменной `start`, а последний — переменной `finish` с помощью `Array.reverse`. Затем мы используем длину массива аргументов, чтобы понять, через сколько городов мы проходим. Количество остановок — это длина массива аргументов за вычетом конечной остановки. Метод обеспечивает гибкость, позволяя использовать функцию `directions` для обработки любого количества остановок.

Как и для массивов, оператор распространения можно использовать для объектов (см. страницу GitHub, посвященную остаточным свойствам и распространению, oreil.ly/kCpEL). В этом примере мы объединим два объекта в третий объект:

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

const dinner = "mac and cheese";

const backpackingMeals = {
  ...morning,
  dinner
};
```

```
console.log(backpackingMeals);

// {
//   breakfast: "oatmeal",
//   lunch: "peanut butter and jelly",
//   dinner: "mac and cheese"
// }
```

Асинхронный JavaScript

Все приведенные ранее в этой главе примеры кода были синхронными — содержали список инструкций, которые выполняются по порядку. Например, чтобы использовать JavaScript для обработки простых манипуляций с DOM, мы напишем:

```
const header = document.getElementById("heading");
header.innerHTML = "Hey!";
```

Это инструкции. «Эй, выбери-ка вон тот элемент с идентификатором `heading`. А как справишься с этим, запиши-ка приветствие в его innerHTML». Код работает синхронно. Пока происходит одна операция, остальной код ждет.

Но в современном интернете необходимо выполнять асинхронные задачи. Для этого часто приходится ждать завершения некоторой работы. Нам может потребоваться доступ к базе данных, потоковая передача видео- или аудиоконтента или данные из API. В JavaScript асинхронные задачи не блокируют основной поток, позволяя коду делать что-то еще, пока он ждет данные от API. За последние несколько лет JavaScript сильно изменился в сторону упрощения обработки асинхронных действий. Рассмотрим некоторые функции, делающие эту обработку возможной.

Простые промисы и функция fetch

Выполнение запроса к REST API раньше было довольно обременительным. Требовалось написать более 20 строк вложенного кода только для того, чтобы загрузить данные в приложение. Затем, благодаря комитету ECMAScript, появилась функция `fetch()`, которая упростила этот процесс.

Попробуем получить данные из API randomuser.me. Этот API содержит такую информацию, как адрес электронной почты, имя, номер телефона, местоположение и прочие данные поддельных участников, и поэтому отлично подходит для использования в качестве фиктивных данных. Функция `fetch` принимает URL-адрес этого ресурса в качестве единственного параметра:

```
console.log(fetch("https://api.randomuser.me/?nat=US&results=1"));
```

В консоли появляется *промис* (обещание), который поможет нам разобраться в асинхронном поведении JavaScript. Промис — это объект, показывающий состояние асинхронной операции: ожидает, завершена или потерпела неудачу. Браузер словно говорит: «Эй, я сделаю все возможное, чтобы получить эти данные. В любом случае я вернусь и расскажу, как все прошло».

Итак, вернемся к результату функции `fetch`. Промис отражает состояние до получения данных. Добавим функцию под названием `.then()`. Эта функция примет функцию обратного вызова, которая будет запущена, если предыдущая операция завершилась успешно. Другими словами, мы получим какие-то данные, а затем сделаем что-нибудь еще.

Еще мы хотим преобразовать ответ в JSON:

```
fetch("https://api.randomuser.me/?nat=US&results=1").then(res =>
  console.log(res.json())
);
```

Метод `then` вызовет функцию обратного вызова, как только промис будет разрешен. Все, что вернется из этой функции, станет аргументом следующей функции `then`. Таким образом, функции для обработки успешно выполненного промиса свяжутся вместе:

```
fetch("https://api.randomuser.me/?nat=US&results=1")
  .then(res => res.json())
  .then(json => json.results)
  .then(console.log)
  .catch(console.error);
```

Сначала мы используем `fetch`, чтобы выполнить GET-запрос к randomuser.me. Если запрос выполнен успешно, мы преобразуем тело ответа в JSON. Затем мы возьмем данные JSON и вернем результаты, а после этого отправим результаты в функцию `console.log`, которая запишет их в консоль. Наконец, функция `catch` выполнит обратный вызов, если выборка не разрешилась успешно. Любая ошибка, возникшая при получении данных из randomuser.me, будет основана на этом обратном вызове. Мы просто запишем ошибку в консоль с помощью `console.error`.

Async и await

Еще один популярный подход к работе с промисами — это создание функции `async`. Некоторые разработчики предпочитают синтаксис асинхронных функций, потому что он выглядит более знакомым и больше похож на код синхронной функции. Вместо того чтобы ждать результатов промиса и его обработки с помощью цепочки функций `then`, функциям `async` можно приказать дождаться

разрешения промиса перед дальнейшим выполнением любого кода, найденного в функции.

Сделаем еще один запрос API, но обернем его функцией `async`:

```
const getFakePerson = async () => {
  let res = await fetch("https://api.randomuser.me/?nat=US&results=1");
  let { results } = res.json();
  console.log(results);
};

getFakePerson();
```

Обратите внимание, что функция `getFakePerson` объявляется с использованием ключевого слова `async`. Это делает функцию асинхронной, и она может дождаться разрешения промисов, прежде чем выполнять код дальше. Ключевое слово `await` используется перед вызовами промисов. Оно приказывает функции дождаться разрешения промиса. Этот код выполняет ту же задачу, что и код в предыдущем разделе, в котором используются функции `then`. Ну или почти.

```
const getFakePerson = async () => {
  try {
    let res = await fetch("https://api.randomuser.me/?nat=US&results=1");
    let { results } = res.json();
    console.log(results);
  } catch (error) {
    console.error(error);
  }
};

getFakePerson();
```

Итак, теперь этот код выполняет ту же задачу, что и код в предыдущем разделе, в котором используются функции `then`. Если вызов `fetch` успешен, результаты будут выведены в консоль. В случае неудачи мы запишем ошибку в консоль с помощью `console.error`. При использовании `async` и `await` вы должны заключить вызов промиса в блок `try...catch` для обработки любых ошибок, которые могут возникнуть из-за неразрешенного промиса.

Сборка промисов

При выполнении асинхронного запроса происходит одно из двух: все работает по плану или возникает ошибка. Возможно множество разных типов успешных или неудачных запросов. Например, мы можем попробовать несколько способов получения данных и в итоге получить несколько типов ошибок. Промисы позволяют упростить решение задачи до «сработало — не сработало».

Функция `getPeople` возвращает новый промис, который делает запрос к API. Если промис выполнен успешно, данные загрузятся. Если он не выполнен, возникнет ошибка:

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

Теперь промис создан, но еще не используется. Мы можем использовать промис, вызвав функцию `getPeople` и передав количество элементов, которое нужно загрузить. Можно добавить функцию `then` для выполнения каких-либо действий после выполнения промиса. Когда промис отклоняется, любые детали передаются обратно в функцию `catch` или блоку `catch` при использовании синтаксиса `async` и `await`:

```
getPeople(5)
  .then(members => console.log(members))
  .catch(error => console.error(`getPeople failed: ${error.message}`))
);
```

Промисы упрощают работу с асинхронными запросами, и это хорошо, потому что в JavaScript постоянно приходится работать с асинхронностью. Современному разработчику на JavaScript необходимо твердое понимание асинхронного поведения.

Классы

До версии ES2015 в спецификации JavaScript не было официального синтаксиса классов. Когда были введены классы, возникло много волнений по поводу того, насколько их синтаксис похож на синтаксис классов в традиционных объектно-ориентированных языках, таких как Java и C++. В последние несколько лет библиотека React сильно опиралась на классы при создании компонентов пользовательского интерфейса. Сегодня React начинает отходить от классов, используя для создания компонентов функции. Но классы по-прежнему будут повсюду, особенно в устаревшем коде React и в мире JavaScript, поэтому мы быстро их рассмотрим.

В JavaScript используется так называемое прототипное наследование. Этот метод можно применить для создания структур, похожих на объектно-ориентированные. Создадим конструктор `Vacation`, вызываемый оператором `new`:

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

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

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

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

Этот код создает нечто похожее на пользовательский тип в объектно-ориентированном языке. У объекта `Vacation` есть свойства (`destination`, `length`) и метод (`print`). Экземпляр `maui` наследует метод `print` через прототип. У тех, кто привык к классической объектной ориентации, подобное положение дел вызывало раздражение. Поэтому в ES2015 появилось объявление класса, слегка успокоившее разработчиков, но на самом деле JavaScript по-прежнему работает так же: функции являются объектами, а наследование осуществляется через прототип. Классы — это просто синтаксический сахар поверх заумного синтаксиса прототипов:

```
class Vacation {
  constructor(destination, length) {
    this.destination = destination;
    this.length = length;
  }

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

Когда вы создаете класс, то пишете его имя с заглавной буквы. Затем вы можете создать новый экземпляр класса, используя ключевое слово `new`. И далее вызвать собственный метод класса:

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

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

Когда объект класса создан, его можно использовать сколько угодно раз для создания новых экземпляров `Vacation`. Классы также можно расширять. Если класс расширяется, подкласс по умолчанию наследует его свойства и методы, после чего вы сможете ими управлять.

`Vacation` можно использовать как абстрактный класс для создания различных типов отпусков. Например, `Expedition` может расширить класс `Vacation`, включив в него снаряжение:

```
class Expedition extends Vacation {
  constructor(destination, length, gear) {
    super(destination, length);
    this.gear = gear;
  }

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

Это простое наследование, при котором подкласс наследует свойства суперкласса. Вызывая метод `print` класса `Vacation`, мы можем добавить новый контент к тому, который печатается в методе `print` класса `Expedition`. Создание нового экземпляра работает точно так же — создайте переменную и используйте ключевое слово `new`:

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

trip.print();

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

Модули ES6

Модуль JavaScript — это фрагмент кода, который можно многократно использовать и включать в другие файлы JavaScript, не вызывая конфликтов переменных. Модули JavaScript хранятся в отдельных файлах, по одному файлу на модуль. Существует два варианта экспорта модуля: экспорт нескольких объектов JavaScript из одного модуля или одного объекта JavaScript для каждого модуля.

В файл `text-helpers.js` экспортируются две функции:

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

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

Ключевое слово `export` можно использовать для экспорта любого типа JavaScript, который будет использоваться в другом модуле. В этом примере экспортируются

функции `print` и `log`. Любые переменные, объявленные в файле text-helpers.js, останутся локальными для этого модуля.

Модули также могут экспортировать одну основную переменную. Для этого можно использовать ключевое слово `export default`. Например, экспортируем в файл mt-freel.js конкретную экспедицию:

```
export default new Expedition("Mt. Freel", 2, ["water", "snack"]);
```

Ключевое слово `export default` заменяет `export`, если с его помощью нужно экспортировать только один тип. И `export`, и `export default` могут использоваться с любыми типами JavaScript: примитивами, объектами, массивами и функциями.

Модули можно использовать в других файлах JavaScript с помощью оператора `import`. Модули с множественным экспортом хорошо работают совместно с деструктуризацией объекта. Использующие экспорт по умолчанию модули импортируются в одну переменную:

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

print("printing a message");
log("logging a message");

freel.print();
```

Можно указать переменные модуля локально под разными именами переменных:

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

p("printing a message");
l("logging a message");
```

Также можно импортировать все в одну переменную, используя знак *:

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

Синтаксис операторов `import` и `export` еще не полностью поддерживается всеми браузерами или Node. Однако, как и любой новый синтаксис JavaScript, он поддерживается транспилятором Babel. Поэтому вы можете использовать эти операторы в своем исходном коде.

CommonJS

CommonJS — это паттерн модуля, который поддерживается всеми версиями Node (см. документацию Node.js по модулям, oreil.ly/CN-gA). Вы можете исполь-

зовать его с Babel и webpack. В CommonJS объекты JavaScript экспортируются с помощью `module.exports`.

Например, в CommonJS мы можем экспортировать функции `print` и `log` как объект:

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

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

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

CommonJS не поддерживает оператор `import` и вместо этого импортирует модули функцией `require`:

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

JavaScript действительно быстро развивается и приспосабливается к растущим требованиям, которые разработчики предъявляют к языку, а браузеры быстро внедряют новые функции. Актуальную информацию об этих процессах можно посмотреть в таблице совместимости ESNext (oreil.ly/rxTcg). Многие из функций, включенных в последний синтаксис JavaScript, в ней присутствуют, потому что они поддерживают методы функционального программирования. В функциональном JavaScript мы можем рассматривать код как набор функций, которые можно объединить в приложения. В следующей главе мы рассмотрим функциональные методы подробнее и обсудим, почему их удобно использовать.

ГЛАВА 3

Функциональное программирование с использованием JavaScript

Вы, вероятно, заметили, что в контексте React часто поднимается тема функционального программирования. Функциональные методы все шире используются в JavaScript, особенно в проектах React.

Возможно, вы уже писали функциональный код на JavaScript, не осознавая этого. Если вам приходилось отображать элементы массива или сокращать их количество, то, значит, вы уже были близки к тому, чтобы стать программистом, использующим функциональные технологии. Методы функционального программирования лежат в основе не только React, но и многих библиотек в ее экосистеме.

Интерес к функциональному программированию появился в 1930-х годах с изобретением *лямбда-исчисления* (λ-исчисления)[1]. Функции стали частью математического анализа с момента его появления в XVII веке. Их можно было отправлять в другие функции в качестве аргументов или возвращать из функций в качестве результатов. Более сложные функции, называемые *функциями высшего порядка*, могли манипулировать функциями и использовать их как аргументы и / или результаты. В 1930-х годах Алонзо Черч в Принстоне изобрел лямбда-исчисление, экспериментируя с функциями высшего порядка.

В конце 1950-х годов Джон Маккарти взял понятия, полученные из λ-исчисления, и применил их к новому языку программирования Lisp. В этом языке была реализована концепция функций высшего порядка и функций, применяемых

[1] Scott D. S. λ-*Calculus: Then & Now* (oreil.ly/k0EpX)

в качестве *элементов* (first-class members) или *объектов* (first-class citizens) *первого класса*. Функция считается элементом первого класса, если она может быть объявлена как переменная и отправлена другим функциям в качестве аргумента. Такая функция может быть даже возвращена из другой функции.

В этой главе мы рассмотрим некоторые ключевые концепции функционального программирования и расскажем, как реализовать функциональные методы с помощью JavaScript.

Что значит «функциональное»?

JavaScript поддерживает функциональное программирование, потому что функции в этом языке являются объектами первого класса. Это означает, что они могут делать то же, что и переменные. В последнем синтаксисе JavaScript есть языковые улучшения, которые могут усилить методы функционального программирования. К ним относятся стрелочные функции, промисы и оператор распространения.

В JavaScript функции могут быть частью данных приложения. Возможно, вы заметили, что можете объявлять функции с ключевыми словами `var`, `let` или `const` так же, как и строки, числа или любые другие переменные:

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

log("In JavaScript, functions are variables");

// Функции в JavaScript являются переменными
```

Можно написать ту же функцию в виде стрелочной функции. Программисты, использующие функциональные технологии, предпочитают множество небольших функций, и синтаксис стрелочных функций значительно упрощает эту работу:

```
const log = message => {
  console.log(message);
};
```

Поскольку функции являются переменными, мы можем добавлять их к объектам:

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

```
    console.log(message);
  }
};

obj.log(obj.message);

// Они, как и переменные, могут добавляться к объектам
```

Обе эти функции делают одно и то же — сохраняют функцию в переменной с именем `log`. Причем объявление второй функции с ключевым словом `const` предотвращает ее перезапись.

В JavaScript также можно добавлять функции в массивы:

```
const messages = [
  "They can be inserted into arrays",
  message => console.log(message),
  "like variables",
  message => console.log(message)
];
messages[1](messages[0]); // Они могут вставляться в массивы
messages[3](messages[2]); // как переменные
```

Функции можно передавать другим функциям в качестве аргументов, как и другие переменные:

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

insideFn(message => console.log(message));

// Они могут отправляться другим функциям в качестве аргументов
```

Они также могут быть возвращены из других функций, как и переменные:

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

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

scream("functions can be returned from other functions");
scream("createScream returns a function");
scream("scream invokes that returned function");

// ФУНКЦИИ МОГУТ ВОЗВРАЩАТЬСЯ ИЗ ДРУГИХ ФУНКЦИЙ!!!
// CREATESCREAM ВОЗВРАЩАЕТ ФУНКЦИЮ!!!
// SCREAM ВЫЗЫВАЕТ ВОЗВРАЩЕННУЮ ФУНКЦИЮ!!!
```

Последние два примера относятся к функциям высшего порядка, которые либо принимают, либо возвращают другие функции. Можно описать ту же функцию высшего порядка `createScream` с помощью стрелок:

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

Если для объявления функции вы используете более одной стрелки — это функция высшего порядка.

Можно сказать, что JavaScript — функциональный язык, поскольку его функции относятся к объектам первого класса. То есть функции являются данными. Их можно сохранять, извлекать или передавать через приложения, как переменные.

Императивное и декларативное программирование

Функциональное программирование является частью более широкой парадигмы *декларативного программирования* — стиля программирования, в структуре приложений которого приоритет отдается описанию того, *что* должно произойти, а не того, *как* это должно быть выполнено.

Чтобы понять суть декларативного программирования, сравним его с *императивным программированием* — стилем, в котором описывается, как достичь результатов с помощью кода. Рассмотрим простую задачу создания строки, удобной для URL. Как правило, для этого можно заменить все пробелы в строке дефисами, поскольку пробелы в URL-адресах недопустимы. Рассмотрим императивный подход к решению этой задачи:

```
const string = "Restaurants in Hanalei";
const urlFriendly = "";

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

console.log(urlFriendly); // "Restaurants-in-Hanalei"
```

В этом примере мы перебираем каждый символ в строке, заменяя пробелы по мере их появления. Структура этой программы описывает лишь то, как

именно задача может быть выполнена. Мы используем цикл `for` и оператор `if` и устанавливаем значения с помощью оператора равенства. Беглый взгляд на код ничего нам не сообщит, поэтому императивным программам требуется множество комментариев.

Теперь посмотрим на декларативный подход к той же задаче:

```
const string = "Restaurants in Hanalei";
const urlFriendly = string.replace(/ /g, "-");

console.log(urlFriendly);
```

Здесь мы используем метод `string.replace` и регулярное выражение для замены всех пробелов дефисами. Использование метода `string.replace` — это способ описания того, что должно произойти: замена пробелов в строке. Подробности алгоритма замены абстрагируются внутри функции `replace`. В декларативной программе сам синтаксис описывает, что должно произойти, а подробности его работы абстрагируются.

Декларативные программы легко читать, потому что сам код описывает, что происходит. Например, прочтите синтаксис в следующем примере. В нем подробно описано, что происходит после загрузки участников (members) из API:

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

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

Декларативный подход легче читать и обсуждать. Подробности того, как реализована каждая из этих функций, абстрагируются. Эти крошечные функции имеют говорящие названия и объединены таким образом, чтобы описать, как данные-участники переходят от загрузки к сохранению и рендерингу, и этот подход не требует большого числа комментариев. По сути декларативное программирование — это создание приложений, которые легче обсуждать и, как следствие, легче масштабировать. Подробнее о парадигме декларативного программирования см. https://ru.wikipedia.org/wiki/Декларативное_программирование.

Теперь рассмотрим задачу построения объектной модели документа, или DOM (www.w3.org/DOM). Императивный подход будет определять, как построена DOM:

```
const target = document.getElementById("target");
const wrapper = document.createElement("div");
const headline = document.createElement("h1");

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

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

Этот код создает элементы, настраивает их и добавляет их в документ. Очень сложно вносить изменения, добавлять функции или масштабировать 10 000 строк кода, если модель DOM построена императивно.

Теперь посмотрим, как мы можем декларативно построить DOM с помощью компонента React:

```
const { render } = ReactDOM;

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

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

React имеет декларативный характер. Здесь компонент `Welcome` описывает DOM, который должен быть выведен на экран. Функция `render` использует инструкции, объявленные в компоненте, для построения DOM, абстрагируясь от деталей того, как именно выполняется рендеринг. Мы ясно видим, что хотим отобразить компонент `Welcome` в элемент с идентификатором `target`.

Функциональные концепции

Теперь, когда вы познакомились с функциональным программированием и с тем, что значат термины «функциональное» или «декларативное», мы перейдем к представлению основных концепций функционального программирования: неизменяемость, чистые функции, преобразование данных, функции высшего порядка и рекурсия.

Неизменяемость

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

Если вам нужно выложить в интернет свое свидетельство о рождении, но вы хотите отредактировать или удалить часть информации, у вас есть два варианта: закрасить часть данных маркером в оригинальном документе или внести изменения в его копию. Второй вариант предпочтительнее, поскольку оригинал останется нетронутым для дальнейшего использования.

Точно так же в приложении работают неизменяемые данные. Вместо того чтобы изменять исходные структуры данных, мы создаем измененные копии этих структур данных и используем их.

Чтобы понять, как работает неизменяемость, посмотрим, что значит «изменять» данные. Рассмотрим объект, представляющий цвет зеленого газона (lawn):

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

Создадим функцию, которая будет оценивать этот цвет, чтобы использовать ее для изменения рейтинга цветового объекта:

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

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

В JavaScript аргументы функции ссылаются на фактические данные. Сейчас мы изменили исходный цветовой объект. (Представьте, что вы поручили некоторой фирме отредактировать скан свого свидетельства о рождении, и получили обратно документ, местами закрашенный черным маркером. А вы надеялись, что фирма догадается сделать копию и вернуть оригинал целым и невредимым.) Перепишем функцию `rateColor` так, чтобы она не повреждала исходный код (объект `color`):

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

console.log(rateColor(color_lawn, 5).rating); // 5
console.log(color_lawn.rating); // 0
```

Мы использовали метод `Object.assign`, чтобы изменить цвет. Метод `Object.assign` — это копировальный аппарат. Он берет пустой объект, копирует в него

цвет и перезаписывает значения на копии. Теперь у нас есть объект с новым цветом, причем старый код не изменился.

Мы можем написать ту же функцию, используя стрелочную функцию и оператор распространения объекта. Функция `rateColor` использует оператор распространения, копируя цвет в новый объект и затем задавая новое значение:

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

Эта версия функции `rateColor` работает аналогично предыдущей. Она обрабатывает цвет как неизменяемый объект, но обходится меньшим числом строк и выглядит чище. Обратите внимание, что мы заключаем возвращенный объект в круглые скобки, поскольку стрелка не может указывать на фигурные скобки.

Рассмотрим массив названий цветов:

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

Мы могли бы создать функцию, которая будет добавлять цвета в этот массив, используя метод `Array.push`:

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

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

Однако метод `Array.push` не является неизменяемым. Функция `addColor` изменяет исходный массив, добавляя к нему еще одно поле. Чтобы массив цветов оставался неизменным, используем вместо него метод `Array.concat`:

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

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

Метод `Array.concat` объединяет массивы: берет новый объект с новым заголовком и добавляет его к копии исходного массива.

Для объединения массивов можно использовать оператор распространения так же, как и для копирования объектов. Вот новый JavaScript-эквивалент предыдущей функции добавления цвета:

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

Эта функция копирует исходный список в новый массив, а затем добавляет в эту копию новый объект, содержащий название цвета. Результат неизменяемый.

Чистые функции

Чистая функция — это функция, которая возвращает значение, вычисленное на основе ее аргументов. Чистые функции принимают по крайней мере один аргумент и всегда возвращают значение или другую функцию. Они не вызывают побочных эффектов, поскольку не изменяют глобальные переменные и состояние приложения. Их аргументы рассматриваются как неизменяемые данные.

Чтобы понять, как работают чистые функции, рассмотрим функцию с побочным эффектом:

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

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

selfEducate();
console.log(frederick);

// {name: "Frederick Douglass", canRead: true, canWrite: true}
```

Функция `selfEducate` не является чистой функцией. Она не принимает аргументы, не возвращает значение или функцию и изменяет переменную `Frederick` за пределами своей области видимости. После вызова функции `selfEducate` за пределами функции возникают побочные эффекты:

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

const selfEducate = person => {
  person.canRead = true;
  person.canWrite = true;
  return person;
};
```

```
console.log(selfEducate(frederick));
console.log(frederick);

// {name: "Frederick Douglass", canRead: true, canWrite: true}
// {name: "Frederick Douglass", canRead: true, canWrite: true}
```

ЧИСТЫЕ ФУНКЦИИ ПРИГОДНЫ ДЛЯ ТЕСТИРОВАНИЯ

Чистые функции по своей природе *пригодны для тестирования*. Они ничего не меняют в своем окружении и, следовательно, не требуют сложной настройки тестирования или обособления. Все, что чистой функции нужно для работы, она получает через аргументы. При тестировании чистой функции вы контролируете аргументы и можете с уверенностью оценить результат.

Дадим этой функции аргумент:

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

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

console.log(selfEducate(frederick));
console.log(frederick);

// {name: "Frederick Douglass", canRead: true, canWrite: true}
// {name: "Frederick Douglass", canRead: false, canWrite: false}
```

Такая версия `selfEducate` является чистой функцией. Она вычисляет значение на основе переданного ей аргумента `person`, возвращает новый объект `person` без изменения аргумента и не имеет побочных эффектов.

Теперь рассмотрим функцию с побочным эффектом, которая изменяет DOM:

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

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

Функция `Header` создает заголовок — элемент с соответствующим текстом — и добавляет его в DOM. Эта функция не является чистой. Она не возвращает функцию или значение и вызывает изменения в DOM.

В React пользовательский интерфейс выражается чистыми функциями. В следующем примере `Header` — чистая функция. Ее можно использовать для создания элементов `h1`, как и в предыдущем примере, но без побочных эффектов изменения DOM. Эта функция создаст элемент `h1`, после чего другая часть приложения сможет использовать этот элемент для изменения DOM:

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

Чистые функции — еще одна основная концепция функционального программирования. Они сделают написание кода намного проще, так как во время работы не повлияют на состояние вашего приложения. При написании функций старайтесь следовать трем правилам:

1. Функция должна принимать хотя бы один аргумент.
2. Функция должна возвращать значение или другую функцию.
3. Функция не должна изменять переданный ей аргумент.

Преобразование данных

Как внести изменения в приложение, если его данные неизменны? Функциональное программирование построено на преобразовании данных из одной формы в другую. Мы будем создавать преобразованные копии с помощью функций. Эти функции сделают код менее императивным и, следовательно, уменьшат его сложность.

Не нужно применять специальные среды, чтобы понять, как создать один набор данных на основе другого. В JavaScript изначально есть необходимые инструменты для решения этой задачи. Чтобы освоить функциональный JavaScript, достаточно научиться использовать две основные функции: `Array.map` и `Array.reduce`.

В этом разделе мы рассмотрим, как эти и некоторые другие базовые функции преобразуют данные из одного типа в другой.

Рассмотрим массив, состоящий из названий гимназий:

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

Можно получить список этих и других строк, разделенных запятыми, используя функцию `Array.join`:

```
console.log(schools.join(", "));

// "Yorktown, Washington & Liberty, Wakefield"
```

`join` — это встроенный метод JavaScript, который можно использовать для извлечения строки с разделителями из массива. Исходный массив при этом не повреждается — `join` просто предлагает другой взгляд на него. Подробности того, как создается эта строка, абстрагированы от программиста.

Чтобы создать функцию, которая создаст новый массив гимназий, названия которых начинаются с буквы «W», можно использовать функцию `Array.filter`:

```
const wSchools = schools.filter(school => school[0] === "W");

console.log(wSchools);
// ["Washington & Liberty", "Wakefield"]
```

`Array.filter` — это встроенная функция JavaScript, которая создает новый массив из исходного массива. Она принимает на вход *предикат* — функцию, которая всегда возвращает логическое значение: истина или ложь. `Array.filter` вызывает этот предикат для каждого элемента в массиве. Затем элементы передаются в предикат в качестве аргументов, и возвращаемые значения используются, чтобы решить, будет ли конкретный элемент добавлен в новый массив. В данном случае `Array.filter` проверяет, начинается ли название каждой гимназии с буквы «W».

Для удаления элемента из массива нужно использовать метод `Array.filter` вместо `Array.pop` или `Array.splice`, так как `Array.filter` неизменяемый. В следующем примере функция `cutSchool` возвращает новые массивы, из которых отфильтрованы определенные названия гимназий:

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

console.log(cutSchool("Washington & Liberty", schools).join(", "));

// "Yorktown, Wakefield"

console.log(schools.join("\n"));

// Yorktown
// Washington & Liberty
// Wakefield
```

Здесь функция `cutSchool` используется для возврата нового массива, не содержащего «Washington & Liberty». Затем `join` создает строки из оставшихся двух названий гимназий. `cutSchool` — это чистая функция. Она берет массив гимназий и название гимназии, которое следует удалить, и возвращает новый массив без этого названия.

Еще одна функция массива, необходимая для функционального программирования, — `Array.map`. Вместо предиката она принимает в качестве аргумента

функцию. Эта функция будет вызываться один раз для каждого элемента в массиве, и все, что она вернет, будет добавлено в новый массив:

```
const highSchools = schools.map(school => `${school} High School`);

console.log(highSchools.join("\n"));

// Yorktown High School
// Washington & Liberty High School
// Wakefield High School

console.log(schools.join("\n"));

// Yorktown
// Washington & Liberty
// Wakefield
```

Здесь функция `map` использована для добавления слов «High School» к названию каждой гимназии. Массив гимназий при этом не изменяется.

В последнем примере мы создали массив строк из массива строк. Функция `map` может создавать массив объектов, значений, массивов и других функций — любого типа JavaScript. Вот пример функции `map`, возвращающей объект для каждой гимназии:

```
const highSchools = schools.map(school => ({ name: school }));

console.log(highSchools);

// [
// { name: "Yorktown" },
// { name: "Washington & Liberty" },
// { name: "Wakefield" }
// ]
```

Массив, содержащий объекты, был создан из массива, содержащего строки.

Чтобы создать чистую функцию, которая изменяет один объект в массиве объектов, также можно использовать функцию `map`. В следующем примере изменим название гимназии «Stratford» на «HB Woodlawn», не изменяя массив `schools`:

```
let schools = [
  { name: "Yorktown" },
  { name: "Stratford" },
  { name: "Washington & Liberty" },
  { name: "Wakefield" }
];

let updatedSchools = editName("Stratford", "HB Woodlawn", schools);

console.log(updatedSchools[1]);  // { name: "HB Woodlawn" }
console.log(schools[1]);         // { name: "Stratford" }
```

`schools` — это массив объектов. Переменная `updatedSchools` вызывает функцию `editName`, и мы отправляем ей название гимназии, которое хотим обновить, новое название и массив `schools`. Мы изменяем новый массив, но не вносим изменений в оригинал:

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

Внутри функции `editName` для создания нового массива объектов на основе исходного массива используется функция `map`. Функцию `editName` можно полностью записать в одну строку. Ниже приведен пример той же функции с использованием сокращенной формы инструкции `if…else`:

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

Чтобы преобразовать массив в объект, используйте метод `Array.map` вместе с методом `Object.keys`, который можно использовать для возврата массива ключей из объекта.

Попробуем преобразовать объект `school` в массив гимназий:

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

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

console.log(schoolArray);

// [
// {
// name: "Yorktown",
// wins: 10
// },
// {
```

```
// name: "Washington & Liberty",
// wins: 2
// },
// {
// name: "Wakefield",
// wins: 5
// }
// ]
```

В этом примере метод `Object.keys` возвращает массив гимназий, и можно применить метод `map` к этому массиву для создания нового массива той же длины. Имя нового объекта будет установлено с помощью ключа, а `wins` установится равным заданному значению.

Итак, мы узнали, что можем преобразовать массив с помощью методов `Array.map` и `Array.filter`. Мы также узнали, что можем преобразовать массив в объект, совместно используя методы `Object.keys` и `Array.map`. Последний инструмент, который нам необходимо иметь в функциональном арсенале, — это возможность преобразовывать массив в примитив и другой объект.

Функции `reduce` и `reduceRight` позволяют преобразовать массив в любое значение, включая число, строку, логическое значение, объект или даже функцию.

Допустим, нам нужно найти максимальное число в массиве чисел. Преобразуем массив в число, например с помощью функции `reduce`:

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

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

console.log("maxAge", maxAge);

// 21 > 0 = true
// 18 > 21 = false
// 42 > 21 = true
// 40 > 42 = false
// 64 > 42 = true
// 63 > 64 = false
// 34 > 64 = false
// maxAge 64
```

Массив `ages` сократился до одного значения максимального возраста — `64`. Функция `reduce` принимает два аргумента: функцию обратного вызова и на-

чальное значение. Начальное значение (начальное максимальное значение) равно `0`. Обратный вызов вызывается один раз для каждого элемента в массиве. При первом вызове этого обратного вызова `age` становится равно `21` — это первое значение в массиве, а `max` равно начальному значению `0`. Обратный вызов возвращает большее из двух чисел — `21`, и оно записывается в `max` для следующей итерации. В каждой итерации новое значение сравнивается с максимальным значением и возвращается большее из двух. Наконец, последнее число в массиве сравнивается и возвращается из предыдущего обратного вызова.

Если мы удалим оператор `console.log` и воспользуемся сокращенной формой инструкции `if…else`, то сможем вычислить максимальное значение в любом массиве чисел с помощью следующего синтаксиса:

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

ARRAY.REDUCERIGHT

Функция `Array.reduceRight` работает так же, как `Array.reduce`, но выполняет задачу с конца массива, а не с начала.

Иногда нужно преобразовать массив в объект. В следующем примере функция `reduce` преобразует массив, содержащий цвета, в хеш:

```
const colors = [
  {
    id: "xekare",
    title: "rad red",
    rating: 3
  },
  {
    id: "jbwsof",
    title: "big blue",
    rating: 2
  },
  {
    id: "prigbj",
    title: "grizzly grey",
    rating: 5
  },
  {
    id: "ryhbhsl",
    title: "banana",
    rating: 1
  }
];

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

```
  return hash;
}, {});

console.log(hashColors);

// {
// "xekare": {
// title:"rad red",
// rating:3
// },
// "jbwsof": {
// title:"big blue",
// rating:2
// },
// "prigbj": {
// title:"grizzly grey",
// rating:5
// },
// "ryhbhsl": {
// title:"banana",
// rating:1
// }
// }
```

В этом примере второй аргумент, переданный функции `reduce`, является пустым объектом. Это начальное значение хеша. Во время каждой итерации функция обратного вызова добавляет в хеш новый ключ, используя скобки, и устанавливает значение для этого ключа в поле `id` массива. Таким образом можно использовать функцию `Array.reduce`, чтобы сократить массив до одного значения — в данном случае до объекта.

Можно также преобразовать массив в другой массив с помощью все той же функции `reduce`. Рассмотрим возможность сокращения массива с повторяющимися значениями до массива уникальных значений:

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

const uniqueColors = colors.reduce(
  (unique, color) =>
    unique.indexOf(color) !== -1 ? unique : [...unique, color],
  []
);

console.log(uniqueColors);

// ["red", "green", "blue"]
```

В этом примере массив цветов, содержащий повторы, сокращается до массива уникальных значений. Второй аргумент, отправленный функции `reduce`, — пустой массив. Он будет начальным значением переменной `distinct`. Если в масси-

ве `distinct` еще не содержится указанный цвет, он будет добавлен. В противном случае он будет пропущен, и вернется текущий массив `distinct`.

Функции `map` и `reduce` — главное оружие функционального программирования, в том числе в JavaScript. Если вы хотите стать опытным инженером по JavaScript, овладейте этими функциями. Способность создавать один набор данных из другого — обязательный навык, который полезен для любой парадигмы программирования.

Функции высшего порядка

Использование *функций высшего порядка* так же важно для функционального программирования. Мы уже говорили о функциях высшего порядка и даже использовали некоторые из них в этой главе. Функции высшего порядка — это функции, которые могут управлять другими функциями: принимать функции в качестве аргументов и/или возвращать функции.

Первая категория функций высшего порядка — это функции, которые ожидают другие функции в качестве аргументов. Например, функции `Array.map`, `Array.filter` и `Array.reduce` принимают функции в качестве аргументов.

Рассмотрим, как можно реализовать функцию высшего порядка. В следующем примере мы создадим функцию обратного вызова `invokeIf`, которая будет проверять условие и вызывать функцию обратного вызова, если оно истинно, и другую функцию обратного вызова, если условие ложно:

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

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

const showUnauthorized = () => console.log("Unauthorized!!!");

invokeIf(true, showWelcome, showUnauthorized); // "Welcome!!!"
invokeIf(false, showWelcome, showUnauthorized); // "Unauthorized!!!"
```

Функции `invokeIf` требуется две функции: для ложного и для истинного случая. Мы передаем ей и `showWelcome`, и `showUnauthorized`. Когда условие истинно, вызывается `showWelcome`. Когда ложно, вызывается `showUnauthorized`.

Функции высшего порядка, возвращающие другие функции, позволяют справиться со сложностями асинхронной работы в JavaScript. Они могут помочь нам в создании функций, которые можно использовать повторно, когда нам это удобно.

Каррирование — это функциональная технология, в которой используются функции высшего порядка.

Ниже приведен пример каррирования. Функция `userLogs` привязана к некоторой информации (имени пользователя) и возвращает функцию, которую можно повторно использовать, когда остальная информация (сообщение) станет доступной. В этом примере ко всем сообщениям в консоли будет добавлено имя пользователя. Обратите внимание, что мы используем функцию `getFakeMembers`, которая возвращает промис из главы 2:

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

const log = userLogs("grandpa23");

log("attempted to load 20 fake members");
getFakeMembers(20).then(
  members => log(`successfully loaded ${members.length} members`),
  error => log("encountered an error loading members")
);

// grandpa23 -> попытка загрузки 20 фиктивных сотрудников
// grandpa23 -> успешно загружены 20 сотрудников

// grandpa23 -> попытка загрузки 20 фиктивных сотрудников
// grandpa23 -> при загрузке сотрудников возникла ошибка
```

`userLogs` — это функция высшего порядка. Функция `log` создается из `userLogs`, и каждый раз, когда функция используется `log`, к сообщению добавляется `grandpa23`.

Рекурсия

Рекурсия — это технология, подразумевающая создание функций, которые вызывают сами себя. Во многих задачах, связанных с циклами, в качестве альтернативы может использоваться рекурсивная функция. Рассмотрим задачу обратного отсчета от 10. Для ее решения можно создать цикл `for` или использовать рекурсивную функцию — в нашем примере `countdown`:

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

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

// 10
// 9
// 8
// 7
// 6
```

```
// 5
// 4
// 3
// 2
// 1
// 0
```

Функция `countdown` ожидает в качестве аргументов число и функцию. Она вызывается с числом `10` и функцией обратного вызова. Когда вызывается функция `countdown`, вызывается обратный вызов, который регистрирует текущее значение. Затем функция `countdown` проверяет значение, чтобы убедиться, что оно больше нуля. Если это так, функция `countdown` вызывает себя с уменьшенным значением. В конце концов, значение станет равным `0`, и `countdown` возвратит его обратно на всем пути стека вызовов.

Рекурсия — это паттерн, который особенно хорошо работает с асинхронными процессами. Функции могут восстанавливать себя по готовности, например, когда становятся доступными данные или когда таймер заканчивает работу.

Функцию `countdown` можно изменить, чтобы отсчет происходил с задержкой. Эта измененная версия функции `countdown` может использоваться для создания таймера с обратным отсчетом:

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

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

В этом примере мы создаем 10-секундный обратный отсчет, первоначально вызывая функцию `countdown` с числом `10` в функции, которая регистрирует обратный отсчет. Функция `countdown` ждет одну секунду перед тем как вызвать себя, таким образом создавая часы.

Рекурсия — хороший метод поиска структур данных. Вы можете использовать рекурсию для перебора вложенных папок до тех пор, пока не будет определена папка, содержащая только файлы. Или для итерации по HTML DOM, пока не найдете элемент, не содержащий дочерних элементов. В следующем примере мы будем использовать рекурсию для глубокого обхода объекта с целью извлечения вложенного значения:

```
const dan = {
  type: "person",
  data: {
```

```
    gender: "male",
    info: {
      id: 22,
      fullname: {
        first: "Dan",
        last: "Deacon"
      }
    }
  }
};
deepPick("type", dan);                          // "person"
deepPick("data.info.fullname.first", dan);   // "Dan"
```

Функция `deepPick` может использоваться для доступа к типу `Dan`, хранящемуся сразу в первом объекте, или для поиска имени `Dan` во вложенных объектах. Передавая строку с точечной нотацией, мы можем указать, где искать значения, вложенные глубоко внутри объекта:

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

Функция `deepPick` либо вернет значение, либо будет вызывать себя, пока не вернет значение. Эта функция разбивает строку полей с точечной нотацией в массив и использует деструктуризацию массива, чтобы отделить первое значение от остальных значений. Если не все значения проверены, `deepPick` вызывает себя с другими данными, позволяя заглянуть в объект на один уровень глубже.

Функция продолжает вызывать себя до тех пор, пока строка полей не перестанет содержать точки (пока поля н закончатся). В этом примере вы увидите, как значения для `first`, `left` и `object[first]` меняются по мере работы функции `deepPick`:

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

// First Iteration
// first = "data"
// remaining.join(".") = "info.fullname.first"
// object[first] = { gender: "male", {info} }

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

// Third Iteration
// first = "fullname"
```

```
// remaining.join("." = "first"
// object[first] = {first: "Dan", last: "Deacon" }

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

Рекурсия — это мощный функциональный метод, который интересно реализовать.

Композиция

В функциональных программах логика разбита на небольшие чистые функции, выполняющие отдельные задачи. Эти мелкие функции нужно собрать вместе: объединить их, вызвать их последовательно или параллельно или объединить их в более крупные функции, пока не сформируется готовое приложение.

Существует много различных реализаций, паттернов и технологий композиции. Возможно, вы знакомы с выстраиванием цепочки. В JavaScript функции можно объединять в цепочку через точечную нотацию, чтобы воздействовать на возвращаемое значение предыдущей функции.

К строкам можно применить метод `replace`. Он вернет шаблонную строку, которая также будет иметь метод замены. Следовательно, можно связать вместе методы замены с точечной нотацией для преобразования строки:

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

console.log(clockTime);

// "03:33:33 PM"
```

В этом примере шаблон представляет собой строку. Привязав метод `replace` к концу шаблонной строки, мы можем заменить часы, минуты, секунды и время суток в строке новыми значениями. Сам шаблон останется неизменным, и его можно будет повторно использовать для создания других показаний часов.

`both` — это функция, которая передает значение через две отдельные функции. Выходные данные `civilianHours` становятся входными данными для `appendAMPM`, благодаря чему мы можем изменить дату, используя обе эти функции сразу:

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

Однако этот синтаксис сложен для понимания и, следовательно, сложен для поддержки и масштабирования. Что произойдет, когда нужно будет передать значение через 20 различных функций?

Более элегантный подход — создать функцию высшего порядка для объединения функций в более крупные функции:

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

both(new Date());
```

Такой подход выглядит намного лучше. Его легко масштабировать, потому что мы можем добавить в него функции в любой момент. Он также позволяет легко изменить порядок составных функций.

Функция-композиция — это функция высшего порядка. Она принимает функции в качестве аргументов и возвращает одно значение:

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

Функция `compose` принимает функции в качестве аргументов и возвращает одну функцию. В этой реализации для преобразования функций-аргументов в массив используется оператор распространения `fns`. Затем возвращается функция `arg`, которая ожидает один аргумент. Когда она вызывается, массив `fns` выстраивается в конвейер, начинающийся с аргумента, который нужно пропустить через функции. Аргумент становится начальным значением для функции `compose`, и при каждой итерации возвращается сокращенная функция обратного вызова. Обратите внимание, что функция обратного вызова принимает два аргумента: композицию и функцию `f`. Каждая функция вызывается с аргументом `compose`, который является результатом вывода предыдущей функции. В конце концов будет вызвана последняя функция и вернется последний результат.

Это простой пример функции `compose`, предназначенный для иллюстрации технологии композиции. Эта функция становится более сложной при обработке более одного аргумента или аргументов, которые не являются функциями.

Соберем все вместе

Теперь, когда мы познакомились с основными концепциями функционального программирования, применим их и создадим небольшое приложение JavaScript.

Наша задача — создать тикающие часы. Они должны отображать часы, минуты, секунды и время суток в формате гражданского времени. Каждое поле должно содержать двузначные числа, то есть к однозначным числам нужно добавить начальные нули. Часы также должны тикать и менять значение каждую секунду.

Рассмотрим императивное решение этой задачи:

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

function logClockTime() {
  // Получение строки показания часов в формате гражданского времени
  let time = getClockTime();

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

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

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

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

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

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

  // Подстановка 0 к показанию секунд, чтобы получалась пара цифр
```

```
  if (time.seconds < 10) {
    time.seconds = "0" + time.seconds;
  }
  // Придание показаниям часов формата строки "hh:mm:ss tt"
  return time.hours + ":" + time.minutes + ":" + time.seconds + " " + time.ampm;
}
```

Это решение работает, а в комментариях описано, что происходит. Однако эти функции большие и сложные. Каждая функция делает много работы. Их сложно понять и поддерживать. Посмотрим, как функциональный подход поможет создать более масштабируемое приложение.

Наша цель — разбить логику приложения на более мелкие части — функции. Каждая функция будет сосредоточена на одной задаче, и мы объединим их в более крупные функции.

Создадим несколько функций, которые дают значения и управляют консолью. Нам понадобятся: функция, которая дает одну секунду; функция, которая дает текущее время; несколько функций, которые регистрируют сообщения на консоли и очищают консоль. В функциональных программах нужно использовать функции вместо значений везде, где это возможно. При необходимости мы вызовем функцию для получения значения:

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

Далее нам понадобятся функции для преобразования данных. Следующие три функции помогут преобразовать объект `Date` в объект, который можно использовать для часов:

- `serializeClockTime`

  Принимает объект времени и возвращает объект, который содержит часы, минуты и секунды.

- `civilianHOurs`

  Принимает объект показания часов и возвращает объект, в котором показание преобразуется в формат гражданского времени.

- `appendAMPM`

  Принимает объект показания часов и добавляет к нему время суток (`AM` или `PM`).

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

```
  seconds: date.getSeconds()
});

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

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

Эти три функции используются для преобразования данных без изменения оригинала. Их аргументы рассматриваются как неизменяемые объекты.

Далее нам понадобятся функции высшего порядка:

- `display`

  Принимает целевую функцию и возвращает функцию, которая передает время в адрес цели. В этом примере целью будет метод `console.log`.

- `formatClock`

  Принимает шаблонную строку — *hh:mm:ss tt* — и использует ее для возврата показания часов, отформатированного по критериям, заданным строкой. То есть эта функция заменяет заполнители показаниями часов, минут, секунд и времени суток.

- `prependZero`

  Принимает ключ объекта в качестве аргумента и добавляет ноль перед значением, хранящемся под ключом этого объекта. Функция получает ключ к указанному полю и добавляет перед значениями ноль, если они меньше 10.

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

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

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

Эти функции высшего порядка будут вызываться для создания функций, которые будут повторно использоваться для форматирования показаний часов каждый тик. И `formatClock`, и `prependZero` будут вызываться один раз, изначально устанавливая шаблон или ключ. Возвращаемые ими внутренние функции будут вызываться каждую секунду для форматирования отображаемого времени.

Теперь, когда у нас есть все необходимые функции, создадим тикающие часы. Используем функцию-композицию из предыдущего раздела:

- `convertToCivilianTime`

  Принимает показания часов в качестве аргумента и преобразует их в формат гражданского времени, используя обе его формы.

- `doubleDigits`

  Принимает показания часов в формате гражданского времени и обеспечивает рендеринг двузначных цифр в часах, минутах и секундах, добавляя нули.

- `startTicking`

  Запускает часы, вызывая функции обратного вызова с интервалом в одну секунду. Обратный вызов состоит из всех перечисленных функций. Каждую секунду консоль очищается, происходит получение текущего времени, его преобразование в формат гражданского времени, форматирование и рендеринг `currentTime`.

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

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

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

```
      ),
      oneSecond()
    );

startTicking();
```

Эта декларативная версия часов дает тот же результат, что и императивная версия. Однако у этого подхода есть несколько преимуществ. Эти функции легко тестировать и использовать повторно. Их можно использовать в будущих часах или других цифровых дисплеях. Программа легко масштабируется. Побочных эффектов нет. Вне самих функций глобальных переменных нет. Ошибки могут возникнуть, но их будет легко найти.

В этой главе мы познакомились с принципами функционального программирования. На протяжении всей книги, обсуждая лучшие практики в React, мы продолжим демонстрировать, как много концепций React основано на функциональных методах. В следующей главе мы погрузимся непосредственно в React, чтобы лучше понять принципы, лежащие в ее основе.

ГЛАВА 4

Как работает React

Итак, мы изучили последний актуальный синтаксис и ознакомились с паттернами функционального программирования, лежащими в основе React. Теперь перейдем к следующему шагу — а именно к нашей цели: изучению работы React. Приступим к написанию реального кода на React.

Работая с React, вы, скорее всего, будете создавать свои приложения с помощью JSX — синтаксиса JavaScript на основе тегов, который очень похож на синтаксис HTML. Мы подробно рассмотрим его в следующей главе и продолжим использовать на протяжении всей книги. Однако, чтобы по-настоящему разобраться в React, вам нужно понять его минимальные единицы — элементы. После элементов мы обсудим компоненты и создадим пользовательские компоненты, из которых состоят другие компоненты и элементы.

Настройка страницы

Для работы с React в браузере необходимо импортировать две библиотеки: React и ReactDOM. React — это библиотека для создания представлений. ReactDOM — это библиотека для рендеринга пользовательского интерфейса в браузере. Обе библиотеки доступны в виде сценариев из распакованного CDN (ссылки приведены в коде ниже). Создадим HTML-документ:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8" />
    <title>React Samples</title>
  </head>
  <body>
    <!-- Target container -->
    <div id="root"></div>
```

```
    <!-- React library & ReactDOM (Development Version)-->
    <script
  src="https://unpkg.com/react@16/umd/react.development.js">
  </script>
    <script
  src="https://unpkg.com/react-dom@16/umd/react-dom.development.js">
  </script>

    <script>
      // Pure React and JavaScript code
    </script>
  </body>
</html>
```

Это минимальные требования для работы с React в браузере. Можно разместить свой сценарий JavaScript в отдельном файле, но он должен быть загружен где-нибудь на странице после загрузки React. Мы собираемся использовать версию React для разработки, чтобы все сообщения об ошибках и предупреждения выводились в консоли браузера. Вы можете использовать уменьшенную релизную версию (react.production.min.js и react.production.min.js), если захотите убрать предупреждения.

Элементы React

HTML — это просто набор инструкций, которые браузер выполняет при создании DOM. Элементы, составляющие HTML-документ, становятся элементами DOM, когда браузер загружает код HTML и отображает пользовательский интерфейс.

Допустим, вам нужно построить иерархию HTML для рецепта. Возможное решение такой задачи может выглядеть примерно так:

```
<section id="baked-salmon">
  <h1>Baked Salmon</h1>
  <ul class="ingredients">
    <li>2 lb salmon</li>
    <li>5 sprigs fresh rosemary</li>
    <li>2 tablespoons olive oil</li>
    <li>2 small lemons</li>
    <li>1 teaspoon kosher salt</li>
    <li>4 cloves of chopped garlic</li>
  </ul>
  <section class="instructions">
    <h2>Cooking Instructions</h2>
    <p>Preheat the oven to 375 degrees.</p>
    <p>Lightly coat aluminum foil with oil.</p>
    <p>Place salmon on foil</p>
```

```
    <p>Cover with rosemary, sliced lemons, chopped garlic.</p>
    <p>Bake for 15-20 minutes until cooked through.</p>
    <p>Remove from oven.</p>
  </section>
</section>
```

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

В прошлом веб-сайты состояли из отдельных HTML-страниц. Когда пользователь перемещался по этим страницам, браузер запрашивал и загружал различные HTML-документы. Изобретение AJAX (асинхронного JavaScript и XML) подарило нам одностраничные приложения, или SPA (https://ru.wikipedia.org/wiki/Одностраничное_приложение). Поскольку браузеры теперь могут с помощью AJAX запрашивать и загружать маленькие фрагменты данных, целые веб-приложения теперь могут работать на одной странице и обновлять пользовательский интерфейс инструментами JavaScript.

В SPA браузер изначально загружает один HTML-документ. Перемещаясь по сайту, пользователи фактически остаются на той же странице. JavaScript уничтожает пользовательский интерфейс и создает новый всякий раз, когда пользователь взаимодействует с приложением. Может показаться, что вы перемещаетесь со страницы на страницу, но на самом деле вы находитесь на одной HTML-странице, а JavaScript выполняет всю работу.

DOM API (https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model) представляет собой набор объектов, которые JavaScript может использовать при взаимодействии с браузером для изменения модели DOM. Если вы когда-либо использовали функцию `document.createElement` или `document.appendChild`, то вы уже работали с DOM API.

React — это библиотека, которая предназначена для обновления DOM браузера. Нам больше не нужно беспокоиться о сложностях, связанных с созданием высокопроизводительных SPA, потому что React делает все сам. В React мы не взаимодействуем напрямую с DOM API. Вместо этого мы сообщаем React, что нам нужно построить, а она сама занимается рендерингом и согласованием элементов, которые нужно создать.

Модель DOM браузера состоит из элементов DOM. Аналогично React DOM состоит из элементов React. Элементы DOM и React на вид похожи, но на самом деле они совершенно разные. Элемент React — это описание того, как должен выглядеть фактический элемент DOM. Другими словами, элементы React — это инструкции по созданию DOM браузера.

Мы можем создать элемент React для представления тега `h1c` с помощью функции `React.createElement`:

```
React.createElement("h1", { id: "recipe-0" }, "Baked Salmon");
```

Первый аргумент определяет тип элемента, который мы хотим создать — в данном случае `h1`. Второй аргумент определяет свойства элемента. Элемент `h1` имеет идентификатор `recipe-0`. Третий аргумент соответствует дочерним элементам: это могут быть любые узлы, вставленные между открывающим и закрывающим тегами (в данном случае только некоторый текст).

Во время рендеринга React преобразует этот элемент в реальный элемент DOM:

```
<h1 id="recipe-0">Baked Salmon</h1>
```

Свойства аналогичным образом применяются к новому элементу DOM: они добавляются к тегу как атрибуты, а дочерний текст добавляется как текст внутри элемента. Элемент React — это просто литерал JavaScript, который сообщает React, как создать элемент DOM.

В журнале этот элемент выглядит так:

```
{
  $$typeof: Symbol(React.element),
  "type": "h1",
  "key": null,
  "ref": null,
  "props": {id: "recipe-0", children: "Baked Salmon"},
  "_owner": null,
  "_store": {}
}
```

Именно так выглядит структура элемента React. Некоторые поля используются самим React: `_owner`, `_store`, и `$$typeof`. Поля `key` и `ref` тоже важны для элементов React, но мы познакомимся с ними позже. А пока подробнее рассмотрим поля `type` и `props`.

Свойство `type` элемента React сообщает React, какой тип элемента HTML или SVG нужно создать. Свойство `props` содержит данные и дочерние элементы, необходимые для создания элемента `DOM`. Свойство `children` предназначено для рендеринга других вложенных элементов в виде текста.

СОЗДАНИЕ ЭЛЕМЕНТОВ

Мы заглядываем в объект, который возвращает функция `React.createElement`. Но создавать подобные элементы вручную не нужно, вместо этого вы всегда будете использовать функцию `React.createElement`.

ReactDOM

Создав элемент React, мы захотим увидеть его в браузере. ReactDOM содержит необходимые для этого инструменты, например метод `render`.

Мы можем отобразить элемент React и его дочерние элементы в DOM с помощью метода `React DOM.render`. Для этого мы передадим этот элемент в качестве первого аргумента, а вторым аргументом будет целевой узел, в котором нужно отобразить элемент:

```
const dish = React.createElement("h1", null, "Baked Salmon");

ReactDOM.render(dish, document.getElementById("root"));
```

Рендеринг элемента `title` в DOM добавит элемент `h1` в `div` с идентификатором `root`, который мы уже определили в нашем HTML. Этот `div` размещается внутри тега `body`:

```
<body>
  <div id="root">
    <h1>Baked Salmon</h1>
  </div>
</body>
```

Все, что связано с рендерингом элементов в DOM, находится в пакете ReactDOM. В версиях React вплоть до React 16 в DOM можно было отображать только один элемент. Сегодня можно также отображать массивы. Когда такая возможность была анонсирована на ReactConf 2017, все аплодировали и кричали. Это было большое дело. Вот как оно выглядит:

```
const dish = React.createElement("h1", null, "Baked Salmon");
const dessert = React.createElement("h2", null, "Coconut Cream Pie");

ReactDOM.render([dish, dessert], document.getElementById("root"));
```

Этот код отобразит оба элемента как одноуровневые внутри корневого контейнера. Надеемся, вы тоже аплодируете и кричите!

В следующем разделе мы узнаем, как использовать метод `props.children`.

Потомки

React отображает дочерние элементы с помощью метода `props.children`. В предыдущем разделе мы отобразили текстовый элемент как дочерний элемента `h1`, и поэтому для метода `props.children` было установлено значение `"Baked Salmon"`. Мы можем отображать другие элементы React как дочерние, создавая

дерево элементов. Вот почему мы используем термин *дерево элементов*: у него есть один корневой элемент, от которого вырастает множество ветвей.

Рассмотрим неупорядоченный список ингредиентов:

```
<ul>
  <li>2 lb salmon</li>
  <li>5 sprigs fresh rosemary</li>
  <li>2 tablespoons olive oil</li>
  <li>2 small lemons</li>
  <li>1 teaspoon kosher salt</li>
  <li>4 cloves of chopped garlic</li>
</ul>
```

В этом примере неупорядоченный список является корневым элементом и имеет шесть дочерних элементов. Мы можем представить этот `ul` и его дочерние элементы с помощью `React.createElement`:

```
React.createElement(
  "ul",
  null,
  React.createElement("li", null, "2 lb salmon"),
  React.createElement("li", null, "5 sprigs fresh rosemary"),
  React.createElement("li", null, "2 tablespoons olive oil"),
  React.createElement("li", null, "2 small lemons"),
  React.createElement("li", null, "1 teaspoon kosher salt"),
  React.createElement("li", null, "4 cloves of chopped garlic")
);
```

Каждый дополнительный аргумент, передаваемый функции `createElement`, является еще одним дочерним элементом. React создает массив этих дочерних элементов и устанавливает для него значение `props.children`.

Проверив получившийся элемент React, мы увидим, что каждый элемент списка представлен элементом React и добавлен в массив `props.children`. Если мы выведем этот элемент в журнал:

```
const list = React.createElement(
  "ul",
  null,
  React.createElement("li", null, "2 lb salmon"),
  React.createElement("li", null, "5 sprigs fresh rosemary"),
  React.createElement("li", null, "2 tablespoons olive oil"),
  React.createElement("li", null, "2 small lemons"),
  React.createElement("li", null, "1 teaspoon kosher salt"),
  React.createElement("li", null, "4 cloves of chopped garlic")
);

console.log(list);
```

то результат будет таким:

```
{
    "type": "ul",
    "props": {
    "children": [
    { "type": "li", "props": { "children": "2 lb salmon" } … },
    { "type": "li", "props": { "children": "5 sprigs fresh rosemary"} … },
    { "type": "li", "props": { "children": "2 tablespoons olive oil" } … },
    { "type": "li", "props": { "children": "2 small lemons"} … },
    { "type": "li", "props": { "children": "1 teaspoon kosher salt"} … },
    { "type": "li", "props": { "children": "4 cloves of chopped garlic"} … }
    ]
    ...
    }
}
```

Теперь видно, что каждый элемент списка является дочерним. Ранее в этой главе мы писали код HTML для всего рецепта, расположив его в элементе `section`. Чтобы повторить это с помощью React, мы будем использовать серию вызовов `createElement`:

```
React.createElement(
  "section",
  { id: "baked-salmon" },
  React.createElement("h1", null, "Baked Salmon"),
  React.createElement(
    "ul",
    { className: "ingredients" },
    React.createElement("li", null, "2 lb salmon"),
    React.createElement("li", null, "5 sprigs fresh rosemary"),
    React.createElement("li", null, "2 tablespoons olive oil"),
    React.createElement("li", null, "2 small lemons"),
    React.createElement("li", null, "1 teaspoon kosher salt"),
    React.createElement("li", null, "4 cloves of chopped garlic")
  ),
  React.createElement(
    "section",
    { className: "instructions" },
    React.createElement("h2", null, "Cooking Instructions"),
    React.createElement("p", null, "Preheat the oven to 375 degrees."),
    React.createElement("p", null, "Lightly coat aluminum foil with oil."),
    React.createElement("p", null, "Place salmon on foil."),
    React.createElement(
      "p",
      null,
      "Cover with rosemary, sliced lemons, chopped garlic."
    ),
    React.createElement(
      "p",
      null,
```

```
      "Bake for 15-20 minutes until cooked through."
    ),
    React.createElement("p", null, "Remove from oven.")
  )
);
```

CLASSNAME В REACT

Любой элемент, имеющий атрибут HTML `class`, использует свойство `className` вместо `class`, поскольку `class` — зарезервированное слово в JavaScript. Вот так выглядит чистый код React. В конечном итоге чистый код React — это то, что работает в браузере. Приложение React — это дерево элементов React, которые происходят из одного корневого элемента. Элементы React — это инструкции, которые React будет использовать для создания пользовательского интерфейса в браузере.

Создание элементов с данными

Основным преимуществом React является его способность отделять данные от элементов пользовательского интерфейса. Поскольку React — это JavaScript, мы можем добавить логику JavaScript, которая поможет нам построить дерево компонентов React. Например, ингредиенты можно хранить в массиве, и мы можем сопоставить массив с элементами React.

Вернемся назад и вспомним о неупорядоченном списке:

```
React.createElement(
  "ul",
  null,
  React.createElement("li", null, "2 lb salmon"),
  React.createElement("li", null, "5 sprigs fresh rosemary"),
  React.createElement("li", null, "2 tablespoons olive oil"),
  React.createElement("li", null, "2 small lemons"),
  React.createElement("li", null, "1 teaspoon kosher salt"),
  React.createElement("li", null, "4 cloves of chopped garlic")
);
```

Данные, используемые в этом списке ингредиентов, можно представить с помощью массива JavaScript:

```
const items = [
  "2 lb salmon",
  "5 sprigs fresh rosemary",
  "2 tablespoons olive oil",
  "2 small lemons",
  "1 teaspoon kosher salt",
  "4 cloves of chopped garlic"
];
```

Из этих данных можно создать список с правильным количеством элементов без необходимости жесткого кодирования каждого из них. Мы можем отображать массив и создавать элементы списка для любого количества ингредиентов:

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

Этот синтаксис создает элемент React для каждого ингредиента в массиве. Каждая строка отображается в дочерних элементах списка как текст. Значение для каждого ингредиента отображается как элемент списка.

При запуске этого кода вы увидите предупреждение консоли, подобное показанному на рис. 4.1.

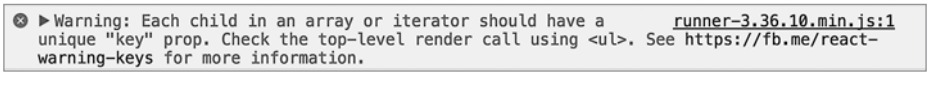


Рис. 4.1. Предупреждение в консоли

Когда мы создаем список дочерних элементов путем итерации по массиву, React хочет, чтобы каждый из них имел свойство `key`. Свойство `key` помогает React эффективно обновлять DOM. Вы можете убрать это предупреждение, добавив уникальное свойство `key` для каждого элемента записи списка. Вы можете использовать индекс массива для каждого ингредиента как уникальное значение:

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

О ключах мы позже поговорим более подробно, когда будем обсуждать JSX, но их добавление к каждому элементу списка уберет предупреждение из консоли.

Компоненты React

Пользовательский интерфейс состоит из отдельных частей, независимо от его размера, содержимого или технологий, используемых для его создания. Кнопки. Списки. Заголовки. Все эти части вместе составляют пользовательский интерфейс.

Рассмотрим приложение, содержащее три разных рецепта. Данные в каждом его поле разные, но части, необходимые для создания рецепта, одинаковы (рис. 4.2).

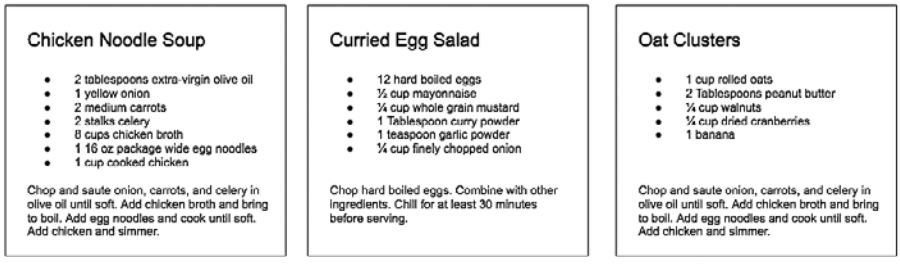


Рис. 4.2. Приложение с рецептами

В React мы описываем каждую из этих частей как *компонент*. Компоненты позволяют повторно использовать одну и ту же структуру, а затем заполнять эти структуры разными наборами данных.

При рассмотрении пользовательского интерфейса, который вы хотите создать с помощью React, ищите возможности разбить элементы на части, которые можно использовать повторно. Например, у рецептов на рис. 4.3 есть названия, списки ингредиентов и инструкции. Все они являются частью более крупного рецепта — компонента приложения. Можно создать компонент для каждой из выделенных частей: ингредиенты, инструкции и так далее.

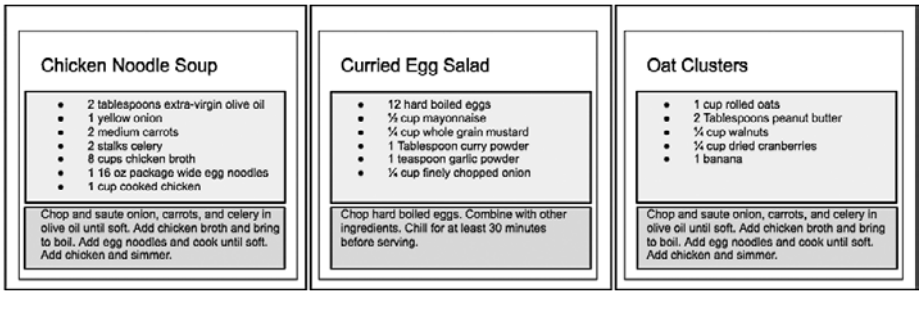


Рис. 4.3. Выделен каждый компонент: приложение, список ингредиентов, инструкции

Подумайте, масштабируется ли такой подход. Если мы хотим отобразить один рецепт, наша структура компонентов позволит сделать это. Если мы хотим отобразить 10 000 рецептов, мы просто создадим 10 000 новых экземпляров компонента.

Теперь создадим компонент с помощью функции. Эта функция вернет повторно используемую часть пользовательского интерфейса. Создадим функцию,

которая возвращает неупорядоченный список ингредиентов. На этот раз мы приготовим десерт с помощью функции `IngredientsList`:

```
function IngredientsList() {
  return React.createElement(
    "ul",
    { className: "ingredients" },
    React.createElement("li", null, "1 cup unsalted butter"),
    React.createElement("li", null, "1 cup crunchy peanut butter"),
    React.createElement("li", null, "1 cup brown sugar"),
    React.createElement("li", null, "1 cup white sugar"),
    React.createElement("li", null, "2 eggs"),
    React.createElement("li", null, "2.5 cups all purpose flour"),
    React.createElement("li", null, "1 teaspoon baking powder"),
    React.createElement("li", null, "0.5 teaspoon salt")
  );
}

ReactDOM.render(
  React.createElement(IngredientsList, null, null),
  document.getElementById("root")
);
```

Имя компонента — `IngredientsList`. Функция выводит элементы следующим образом:

```
<IngredientsList>
  <ul className="ingredients">
    <li>1 cup unsalted butter</li>
    <li>1 cup crunchy peanut butter</li>
    <li>1 cup brown sugar</li>
    <li>1 cup white sugar</li>
    <li>2 eggs</li>
    <li>2.5 cups all purpose flour</li>
    <li>1 teaspoon baking powder</li>
    <li>0.5 teaspoon salt</li>
  </ul>
</IngredientsList>
```

Получилось хорошо, но мы жестко запрограммировали эти данные в компонент. Что, если бы мы могли построить один компонент, а затем передать данные в этот компонент как свойства? А что, если этот компонент сможет отображать данные динамически? Появится ли когда-нибудь такая возможность?

Да, она появилась. Вот набор секретных ингредиентов, необходимых для составления рецепта:

```
const secretIngredients = [
  "1 cup unsalted butter",
  "1 cup crunchy peanut butter",
```

```
  "1 cup brown sugar",
  "1 cup white sugar",
  "2 eggs",
  "2.5 cups all purpose flour",
  "1 teaspoon baking powder",
  "0.5 teaspoon salt"
];
```

Теперь настроим компонент `IngredientsList` для рендеринга, составив из любого количества элементов `items` список `li`:

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

Передадим `secretIngridients` как свойство под названием `items`, которое будет вторым аргументом, используемым в `createElement`:

```
ReactDOM.render(
  React.createElement(IngredientsList, { items: secretIngredients }, null),
  document.getElementById("root")
);
```

Посмотрим на DOM. Свойства данных `items` представляют собой массив с восемью ингредиентами. Создав теги `li` с помощью цикла, мы добавили уникальные ключи, используя индекс цикла:

```
<IngredientsList items="[...]">
  <ul className="ingredients">
    <li key="0">1 cup unsalted butter</li>
    <li key="1">1 cup crunchy peanut butter</li>
    <li key="2">1 cup brown sugar</li>
    <li key="3">1 cup white sugar</li>
    <li key="4">2 eggs</li>
    <li key="5">2.5 cups all purpose flour</li>
    <li key="6">1 teaspoon baking powder</li>
    <li key="7">0.5 teaspoon salt</li>
  </ul>
</IngredientsList>
```

Такое создание компонента сделает его более гибким. Независимо от того, состоит ли массив `items` из одного элемента или из ста элементов, компонент будет отображать каждый элемент как элемент списка.

Еще одна корректировка, которую мы можем сделать, — это добавление ссылки на массив `items` из `props` React. Вместо рендеринга `items` мы сделаем элементы доступными для объекта `props`. Начнем с передачи `props` в функцию, затем выполним рендеринг `props.items`:

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

Также можно немного очистить код, деструктурируя элементы из `props`:

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

Все, что связано с пользовательским интерфейсом для `IngredientsList`, инкапсулируется в один компонент. Все нужное собрано в одном месте.

Компоненты React: историческая справка

До появления функциональных компонентов существовали и другие способы создания компонентов. Мы кратко обсудим эти подходы, чтобы вы смогли работать с API в устаревших кодовых базах. Давайте совершим небольшое путешествие во времени по React API прошлых лет.

Остановка 1: createClass

Когда в 2013 году React стал библиотекой с открытым исходным кодом, существовал лишь один способ создать компонент — с помощью функции `createClass`. Использование функции `React.createClass` для создания компонента выглядит так:

```
const IngredientsList = React.createClass({
  displayName: "IngredientsList",
  render() {
```

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

Компоненты, использующие `createClass`, имели метод `render()`, описывающий элементы React, которые должны быть возвращены и обработаны. Суть компонента была такой же: он описывал элемент пользовательского интерфейса, который нужно отобразить.

В версии 15.5 (апрель 2017 года) React выдавал предупреждения при использовании функции `createClass`. В версии 16 (сентябрь 2017 года) функция `React.createClass` официально устарела и была перемещена в отдельный пакет под названием create-react-class.

Остановка 2: компоненты класса

Когда синтаксис класса был добавлен в JavaScript на ES 2015, в React появился новый метод для создания компонентов React. API `React.Component` позволил использовать синтаксис класса для создания нового экземпляра компонента:

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

Сейчас тоже можно создавать компоненты React с использованием синтаксиса класса, но имейте в виду, что `React.Component` постепенно устаревает. Этот синтаксис все еще поддерживается, но вскоре его ждет судьба его товарища `React.createClass`. Отнеситесь к нему как к другу детства, с которым вы много чего пережили, но потом ваши пути разошлись. С этого момента в этой книге для создания компонентов мы будем использовать функции и лишь кратко будем ссылаться на старые паттерны для ознакомления.

ГЛАВА 5

React и JSX

В предыдущей главе мы поговорили о том, как работает React, разбив наши приложения React на небольшие части для многократного использования, называемые компонентами. Эти компоненты отображают деревья элементов и других компонентов. Функция `createElement` позволяет увидеть, как работает React, но разработчики React не должны составлять сложные, трудночитаемые деревья синтаксиса JavaScript. Для эффективной работы с React мы будем использовать JSX.

Аббревиатура JSX включает JS из JavaScript и X из XML. JSX — это расширение JavaScript, которое позволяет определять элементы React с использованием синтаксиса тегов непосредственно в коде JavaScript. Иногда JSX путают с похожим на него HTML. JSX — это еще один способ создания элементов React, который избавляет нас от поиска пропавшей запятой в сложном вызове `createElement`.

В этой главе мы обсудим, как использовать JSX для создания приложения React.

Представление элементов React в JSX

Команда Facebook выпустила JSX с целью получить краткий синтаксис для создания сложных деревьев DOM с атрибутами и сделать React таким же читабельным, как HTML и XML. В JSX тип элемента указывается с помощью тега, атрибуты тега задают свойства, а дочерние элементы добавляются между открывающим и закрывающим тегами.

В качестве дочерних элементов можно использовать другие элементы JSX. Если у вас есть неупорядоченный список, вы можете добавить в него элементы дочернего списка с помощью тегов, как в HTML:

```
<ul>
  <li>1 lb Salmon</li>
  <li>1 cup Pine Nuts</li>
```

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

JSX также работает с компонентами, которые нужно определить через имя класса. На рис. 5.1 массив ингредиентов передается компоненту `IngredientsList` в качестве свойства с помощью JSX.

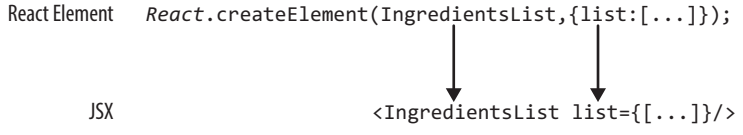


Рис. 5.1. Создание списка IngredientsList с помощью JSX

Мы передали значение компоненту JavaScript в качестве свойства в форме *выражения* JavaScript. Свойства компонента могут быть представлены либо строкой, либо выражением JavaScript. Выражение может включать массивы, объекты и даже функции и должно быть заключено в фигурные скобки.

Советы по использованию JSX

JSX может показаться интуитивно понятным, и большинство его правил дает синтаксис, аналогичный HTML. Однако при работе с JSX следует учесть несколько моментов.

Вложенные компоненты

JSX позволяет добавлять компоненты как дочерние по отношению к другим компонентам. Например, внутри `IngredientsList` мы можем несколько раз отобразить другой компонент под названием `Ingredient`:

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

className

Поскольку `class` — это зарезервированное слово в JavaScript, для определения атрибута `class` используется `className`:

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

Выражения JavaScript

Выражения JavaScript заключаются в фигурные скобки и указывают, где должны быть вычислены переменные и куда будут возвращены результаты этих вычислений. Например, чтобы отобразить в элементе значение свойства `title`, мы можем вставить это значение в выражение JavaScript. Переменная будет вычислена, а ее значение — возвращено:

```
<h1>{title}</h1>
```

Значения типов, отличных от строки, также должны быть представлены выражениями JavaScript:

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

Вычисление

Код JavaScript, заключенный в фигурные скобки, будет вычислен. То есть к нему будут применены такие операции, как конкатенация или сложение, а найденные в выражениях функции будут вызваны:

```
<h1>{"Hello" + title}</h1>

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

Рендеринг массивов с помощью JSX

JSX — это JavaScript, поэтому вы можете включать JSX непосредственно в функции JavaScript. Например, сопоставить массив с элементами JSX:

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

JSX выглядит понятным и легко читаемым, но его не может интерпретировать браузер. Весь JSX нужно преобразовать в вызовы `createElement`. К счастью, для этой задачи есть отличный инструмент — Babel.

Babel

Многие языки программирования подразумевают транспиляцию исходного кода. JavaScript — это интерпретируемый язык: браузер интерпретирует код как текст, поэтому необходимости компилировать JavaScript нет. Однако не все браузеры поддерживают новейший синтаксис JavaScript, и ни один браузер

не поддерживает синтаксис JSX. Поскольку мы хотим использовать новейшие функции JavaScript вместе с JSX, нам нужна возможность преобразовывать новомодный исходный код во что-то, что браузер сможет интерпретировать. Этот процесс называется транспиляцией, и именно этой задачей занимается Babel (https://babeljs.io/).

Первая версия транспилятора называлась 6to5 и была выпущена в сентябре 2014 года. 6to5 был инструментом для преобразования синтаксиса ES6 в синтаксис ES5, который на тот момент лучше поддерживался веб-браузерами. По мере своего развития транспилятор стал платформой для поддержки всех последних изменений в ECMAScript. К тому же в нем появилась поддержка преобразования JSX в JavaScript. В феврале 2015 года этот транспилятор был переименован в Babel.

Babel используется в продакшене Facebook, Netflix, PayPal, Airbnb и других кампаний. Ранее команда Facebook создала преобразователь JSX по своим стандартам, но вскоре отказалась от него в пользу Babel.

Работать с Babel можно разными способами. Самый простой — включить ссылку на CDN Babel непосредственно в HTML-код, который будет транспилировать любой код в блоках сценария, имеющих тип text/babel. Babel транспилирует исходный код на клиенте перед его запуском. Это не лучшее решение для продакшена, но для начала работы с JSX оно подходит отлично:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8" />
    <title>React Examples</title>
  </head>
  <body>
    <div id="root"></div>

    <!—- Библиотеки React и React DOM -->
    <script
  src="https://unpkg.com/react@16.8.6/umd/react.development.js">
  </script>
    <script
  src="https://unpkg.com/react-dom@16.8.6/umd/react-dom.development.js">
  </script>
    <script
  src="https://unpkg.com/@babel/standalone/babel.min.js">
  </script>

    <script type="text/babel">
      // Здесь размещается код JSX или ссылка на отдельный файл JavaScript,
      // содержащий код JSX
    </script>
  </body>
</html>
```

ПРЕДУПРЕЖДЕНИЕ В КОНСОЛИ ПРИ РАБОТЕ С БРАУЗЕРНОЙ ВЕРСИЕЙ BABEL

При использовании транспилятора в браузере вы увидите предупреждение о необходимости предварительной транспиляции сценариев для продакшена. Не беспокойтесь об этом предупреждении при работе с небольшими примерами. Позже в этой главе мы перейдем на подходящий для продакшена Babel.

Приложение с рецептами в виде JSX

JSX элегантно и понятно выражает элементы React в коде. Недостатком JSX является отсутствие поддержки браузером. Чтобы код был интерпретирован браузером, его необходимо преобразовать из JSX в JavaScript.

Приведенный ниже массив данных содержит два рецепта, которые представляют текущее состояние приложения:

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

```
      "Top them with lettuce, tomatoes, and cheese."
    ]
  }
];
```

Данные представлены в виде массива из двух объектов JavaScript. Каждый объект содержит название рецепта, список ингредиентов и список шагов приготовления блюд.

Создадим пользовательский интерфейс с двумя компонентами: `Menu` — для перечисления рецептов и `Recipe` — для описания пользовательских интерфейсов каждого рецепта. Именно компонент `Menu` мы визуализируем в DOM. Передадим компоненту `Menu` данные в виде свойства `recipes`:

```
// Данные, массив объектов Recipes
const data = [ ... ];

// Функциональный компонент, предназначенный для Recipes
function Recipe (props) {
  ...
}

// Функциональный компонент, предназначенный для Menu
function Menu (props) {
  ...
}
// Вызов ReactDOM.render для рендеринга Menu в текущей DOM-модели
ReactDOM.render(
  <Menu recipes={data} title="Delicious Recipes" />,
  document.getElementById("root")
);
```

Элементы React в компоненте `Menu` выражены в виде JSX. Все содержится в элементе `article`. Элементы `header`, `h1` и `div.recipes` используются для описания DOM меню. Значение свойства `title` будет отображаться в виде текста внутри `h1`:

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

Добавим к элементу `div.recipes` компонент для каждого рецепта:

```
<div className="recipes">
  {props.recipes.map((recipe, i) => (
```

```
    <Recipe
      key={i}
      name={recipe.name}
      ingredients={recipe.ingredients}
      steps={recipe.steps}
    />
  ))}
</div>
```

Чтобы перечислить рецепты в элементе `div.recipes`, мы используем фигурные скобки для добавления выражения JavaScript, которое вернет массив дочерних элементов. Мы можем использовать функцию `map` в массиве `props.recipes`, чтобы вернуть компонент для каждого объекта в массиве. Как упоминалось ранее, каждый рецепт содержит название, ингредиенты и инструкции по приготовлению (шаги). Нам нужно передать эти данные каждому компоненту `Recipe` в виде свойств. Не забудьте использовать свойство `key` для уникальной идентификации каждого элемента.

Код можно улучшить с помощью оператора распространения JSX, который работает как оператор распространения объекта. Он добавит каждое поле объекта `recipe` в качестве свойства компоненту `Recipe`. Приведенный ниже синтаксис передаст компоненту все свойства:

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

Помните, что все свойства передаются компоненту `Recipe`, и, возможно, свойств окажется слишком много.

Еще одно место, где можно улучшить синтаксис компонента `Menu`, — получение аргумента `props`. Чтобы ввести переменные в область видимости этой функции, применим деструктуризацию объекта. Так мы сможем напрямую обращаться к переменным `title` и `recipes`, не добавляя к ним префикса `props`:

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

Запрограммируем компонент для каждого отдельного рецепта:

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

У каждого рецепта есть строка названия, массив объектов для ингредиентов и массив строк для шагов приготовления. Используя деструктуризацию объекта, мы можем указать компоненту локально охватить эти поля по имени, чтобы получить к ним доступ без использования `props.name`, `props.ingredients` или `props.steps`.

Первое показанное здесь выражение JavaScript используется для установки атрибута `id` для корневого элемента `section`. Оно преобразует название рецепта в строку, состоящую из символов нижнего регистра, и заменяет в ней все пробелы дефисами. В результате «Baked Salmon» превращается в «baked-salmon», прежде чем использоваться как атрибут `id` в пользовательском интерфейсе. Значение `name` также отображается в `h1` в качестве текстового узла.

Внутри неупорядоченного списка находится выражение JavaScript. Оно сопоставляет каждый ингредиент с элементом `li`, который выводит название ингредиента. В разделе инструкций используется тот же паттерн для возврата элемента абзаца, в котором отображается каждый шаг. Функции `map` возвращают массивы дочерних элементов.

Полный код приложения должен выглядеть так:

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

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

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

function Menu({ title, recipes }) {
  return (
    <article>
      <header>
        <h1>{title}</h1>
      </header>
```

```
      <div className="recipes">
        {recipes.map((recipe, i) => (
          <Recipe key={i} {...recipe} />
        ))}
      </div>
    </article>
  );
}

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

Когда мы запускаем этот код в браузере, React создает пользовательский интерфейс, используя наши инструкции с данными рецепта (рис. 5.2).

Если вы используете Google Chrome и у вас установлено расширение React Developer Tools Extension, вы можете посмотреть на текущее состояние дерева компонентов. Для этого откройте инструменты разработчика и выберите вкладку Components (рис. 5.3).

Вы увидите объект `Menu` и его дочерние элементы. Массив `data` содержит два объекта для рецептов, и есть два элемента `Recipe`. Каждый элемент имеет свойства — название рецепта, ингредиенты и шаги приготовления. Ингредиенты и шаги передаются своим собственным компонентам в виде данных.

Компоненты создаются на основе данных приложения, передаваемых компоненту `Menu` в качестве свойств. Если мы изменим массив `recipes` и повторно отобразим компонент `Menu`, React изменит DOM настолько эффективно, насколько сможет.

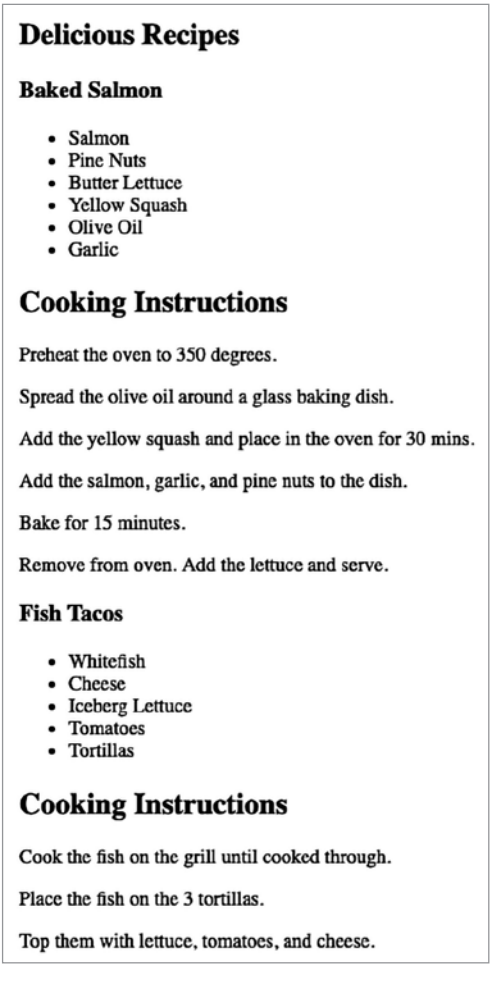


Рис. 5.2. Результат работы приложения Delicious Recipes

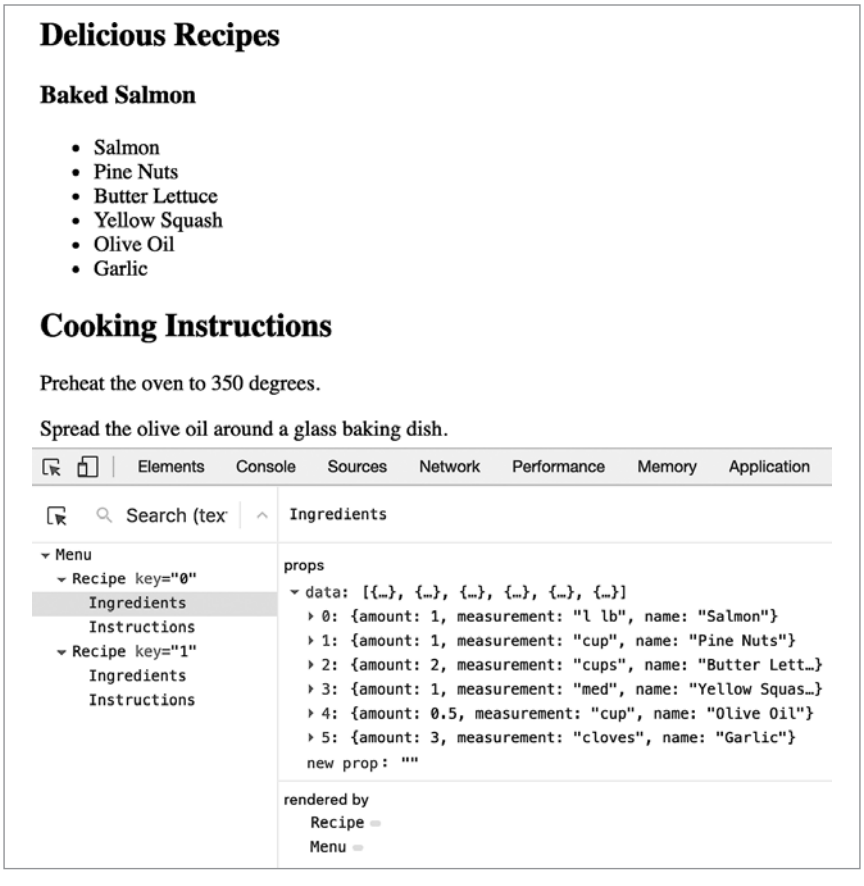


Рис. 5.3. Получение виртуального DOM в React Developer Tools

Фрагменты React

В предыдущем разделе мы отобразили компонент `Menu` — родительский компонент, который, в свою очередь, отображал компонент `Recipe`. Воспользуемся моментом и посмотрим на небольшой пример рендеринга двух родственных компонентов с использованием фрагмента React. Начнем с создания нового компонента под названием `Cat`, который мы отобразим в DOM в `root`:

```
function Cat({ name }) {
  return <h1>The cat's name is {name}</h1>;
}

ReactDOM.render(<Cat name="Jungle" />, document.getElementById("root"));
```

Как и ожидалось, мы получили на выходе `h1`, но что произойдет, если мы добавим тег `p` в компонент `Cat` на том же уровне, что и `h1`?

```
function Cat({ name }) {
  return (
    <h1>The cat's name is {name}</h1>
    <p>He’s good.</p>
  );
}
```

Мы увидим в консоли ошибку, которая сообщит, что смежные элементы JSX должны быть заключены в тег, и порекомендует использовать фрагмент. Сейчас на сцену выйдут фрагменты! React не отображает два или более смежных или родственных элемента как компонент, поэтому мы добавили к ним закрывающий тег вроде `div`. Однако это привело к созданию множества нежелательных тегов и оберток. Используя фрагмент React, сымитируем поведение оболочки, фактически не создавая новый тег.

Начнем с обертывания смежных тегов `h1` и `p` тегом `React.Fragment`:

```
function Cat({ name }) {
  return (
    <React.Fragment>
      <h1>The cat's name is {name}</h1>
      <p>He's good.</p>
    </React.Fragment>
  );
}
```

Предупреждение исчезло. Чтобы повысить удобочитаемость, можно сократить фрагмент:

```
function Cat({ name }) {
  return (
    <>
      <h1>The cat's name is {name}</h1>
      <p>He’s good.</p>
    </>
  );
}
```

Если вы посмотрите на DOM, то не увидите фрагмент в получившемся дереве:

```
<div id="root">
  <h1>The cat's name is Jungle</h1>
  <p>He's good</p>
</div>
```

Фрагменты — это относительно новая функция React, которая устраняет необходимость в дополнительных тегах-оболочках, засоряющих DOM.

Введение в webpack

В начале работы с React в реальных проектах у вас возникнет множество вопросов. Как работать с преобразованием JSX и ESNext? Как управлять зависимостями? Как оптимизировать изображения и CSS?

Для ответов на эти вопросы есть множество инструментов, таких как Browserify, gulp, Grunt, Prepack и другие. Благодаря своему функционалу и широкому распространению среди крупных компаний, одним из ведущих инструментов для создания пакетов стал webpack.

Экосистема React серьезно выросла и включает такие инструменты, как create-react-app, Gatsby и Code Sandbox. Они скрывают многие подробности сборки кода. В оставшейся части этой главы мы настроим свою сборку веб-пакета. Нам достаточно понимать, что код JavaScript / React собирается чем-то вроде webpack, а вот знать, *как* именно это происходит, не столь важно. Можете пропустить этот раздел.

Webpack позиционируется как сборщик модулей, который берет все файлы проекта (JavaScript, LESS, CSS, JSX, ESNext и т. д.) и превращает их в один файл. Два основных преимущества такого объединения — это *модульность* и *сетевая производительность*.

Модульность позволяет разбить исходный код на части или модули, с которыми легче работать в команде.

Сетевая производительность достигается за счет того, что в браузер загружается только одна вещь: пакет. Каждый тег `script` выполняет HTTP-запрос, и для каждого HTTP-запроса существует штраф за задержку. Объединение всех зависимостей в один файл позволяет загружать весь пакет с помощью одного HTTP-запроса, предотвращая лишние задержки.

Помимо сборки кода webpack выполняет следующие задачи:

Разбиение кода

Разбивает код на части, иногда называемые *свертками*, или *слоями*, которые можно загружать по мере необходимости для разных страниц или устройств.

Минификация

Удаляет пробелы, разрывы строк, длинные имена переменных и ненужный код для уменьшения размера файла.

Прогон функции

Отправляет код в одну или несколько — но не во все — среды при тестировании.

Горячая замена модуля (HMR, hot module replacement)

Следит за изменениями в исходном коде. Мгновенно заменяет только обновленные модули.

Приложение рецептов, которое мы создали ранее в этой главе, имеет некоторые ограничения, которые webpack поможет устранить. Использование инструмента вроде webpack для статической сборки клиентского JavaScript позволяет нескольким командам вместе работать над крупномасштабными веб-приложениями. Сборщик модулей webpack также дает дополнительные преимущества:

Модульность

Использование модульного паттерна для экспорта модулей, которые позже будут импортированы или потребуются другой части приложения, делает исходный код более доступным. Это позволяет командам разработчиков работать вместе, каждый в своих файлах, которые затем будут статически объединены в один файл перед отправкой в продакшен.

Составление компонентов

С помощью модулей можно создавать небольшие, простые, повторно используемые компоненты React, которые можно эффективно компоновать в приложения. Компоненты меньшего размера легче понимать, тестировать и использовать повторно. Их также легче заменить при усовершенствовании приложений.

Скорость

Упаковка всех модулей и зависимостей приложения в единый пакет сокращает время загрузки приложения: клиенту нужно сделать всего один HTTP-запрос без ожидания, вызываемого отдельными запросами. Минификация кода в пакете также снижает время загрузки.

Согласованность

Поскольку webpack позволяет собирать JSX и JavaScript, мы можем начать использовать синтаксис JavaScript будущего уже сегодня. Babel поддержи-

вает широкий спектр синтаксиса ESNext, а поэтому нам не придется думать о том, поддерживает ли браузер наш код.

Создание проекта

Чтобы продемонстрировать проект React с нуля, создадим новую папку под названием recipes-app:

```
mkdir recipes-app
cd recipes-app
```

В этом проекте мы выполним следующие шаги:

1. Создадим проект.
2. Разобьем приложение рецептов на компоненты, которые будут находиться в разных файлах.
3. Настроим сборку веб-пакета с использованием Babel.

CREATE-REACT-APP

Для автоматического создания проекта React можно использовать инструмент Create React App. Но сначала мы рассмотрим подробнее, что происходит за кулисами, а потом будем применять готовый инструмент.

1. Создадим проект

Создадим проект и файл package.json с помощью npm, отправив флаг `-y` для установки всех значений по умолчанию. Также установим `webpack`, `webpack-cli`, `react` и `react-dom`:

```
npm init -y
npm install React-dom serve
```

Если мы используем npm 5, флаг `--save` при установке отправлять не нужно. Создадим структуру каталогов для размещения компонентов:

```
recipes-app (folder)
  > node_modules (already added with npm install command)
  > package.json (already added with npm init command)
  > package-lock.json (already added with npm init command)
  > index.html
  > /src (folder)
    > index.js
    > /data (folder)
      > recipes.json
```

```
    > /components (folder)
      > Recipe.js
      > Instructions.js
      > Ingredients.js
```

ОРГАНИЗАЦИЯ ФАЙЛОВ

Нет универсального способа организовать файлы в проекте React. Мы покажем одну из возможных стратегий.

2. Разобьем компоненты на модули

Компонент `Recipe` уже сейчас делает много работы. Он отображает название рецепта, строит неупорядоченный список ингредиентов и показывает инструкции, причем каждый шаг рецепта — как отдельный абзац. Этот компонент должен быть помещен в файл Recipe.js. В любом файле, в котором мы используем JSX, нужно импортировать React в верхней части:

```
// ./src/components/Recipe.js

import React from "react";

export default function Recipe({ name, ingredients, steps }) {
  return (
    <section id="baked-salmon">
      <h1>{name}</h1>
      <ul className="ingredients">
        {ingredients.map((ingredient, i) => (
          <li key={i}>{ingredient.name}</li>
        ))}
      </ul>
      <section className="instructions">
        <h2>Cooking Instructions</h2>
        {steps.map((step, i) => (
          <p key={i}>{step}</p>
        ))}
      </section>
    </section>
  );
}
```

Более функциональный подход к компоненту `Recipe` — разбить его на более мелкие, более сфокусированные функциональные компоненты и собрать их вместе. Начнем с извлечения инструкций в отдельный компонент и создания модуля в отдельном файле, который мы сможем использовать для выполнения любого набора инструкций.

В новом файле с именем Instructions.js создадим следующий компонент:

```
// ./src/components/Instructions.js

import React from "react";

export default function Instructions({ title, steps }) {
  return (
    <section className="instructions">
      <h2>{title}</h2>
      {steps.map((s, i) => (
        <p key={i}>{s}</p>
      ))}
    </section>
  );
}
```

Мы создали новый компонент `Instructions`. Передадим ему заголовок инструкции и шаги приготовления. Таким образом, мы сможем повторно использовать этот компонент для «Инструкций по приготовлению», «Инструкций по выпечке», «Инструкций по подготовке» или «Проверок перед приготовлением» — всего, что можно разбить на шаги.

Теперь подумаем об ингредиентах. В компоненте `Recipe` мы отображаем только названия ингредиентов, но у каждого ингредиента в данных рецепта есть еще количество и размер, для рендеринга которых мы создадим компонент под названием `Ingredient`:

```
// ./src/components/Ingredient.js

import React from "react";

export default function Ingredient({ amount, measurement, name }) {
  return (
    <li>
      {amount} {measurement} {name}
    </li>
  );
}
```

Здесь мы предполагаем, что у каждого ингредиента есть количество, размер и название. Мы деструктурируем эти значения из объекта `props` и отображаем каждое из них в независимых элементах `span`, имеющих атрибут `class`.

Используя компонент `Ingredient`, создадим компонент `IngredientsList`, который мы сможем применять при необходимости отобразить список ингредиентов:

```
// ./src/components/IngredientsList.js

import React from "react";
import Ingredient from "./Ingredient";

export default function IngredientsList({ list }) {
  return (
    <ul className="ingredients">
      {list.map((ingredient, i) => (
        <Ingredient key={i} {...ingredient} />
      ))}
    </ul>
  );
}
```

В этом файле сначала импортируем компонент `Ingredient`, чтобы использовать его для каждого ингредиента. Ингредиенты передаются этому компоненту в виде массива в свойстве под названием `list`. Каждый ингредиент в массиве `list` будет сопоставлен с компонентом `Ingredient`. Оператор распространения JSX передаст все данные в компонент `Ingredient` в качестве свойств.

Использование оператора `spread`:

```
<Ingredient {...ingredient} />
```

это еще один способ выразить:

```
<Ingredient
  amount={ingredient.amount}
  measurement={ingredient.measurement}
  name={ingredient.name}
/>
```

То есть, имея `Ingredient` с такими полями:

```
let ingredient = {
  amount: 1,
  measurement: "cup",
  name: "sugar"
};
```

мы получим:

```
<Ingredient amount={1} measurement="cup" name="sugar" />
```

Теперь, когда у нас есть компоненты для ингредиентов и инструкций, мы можем составлять из них рецепты:

```
// ./src/components/Recipe.js

import React from "react";
import IngredientsList from "./IngredientsList";
import Instructions from "./Instructions";

function Recipe({ name, ingredients, steps }) {
  return (
    <section id={name.toLowerCase().replace(/ /g, "-")}>
      <h1>{name}</h1>
      <IngredientsList list={ingredients} />
      <Instructions title="Cooking Instructions" steps={steps} />
    </section>
  );
}

export default Recipe;
```

Сначала импортируем компоненты, которые собираемся использовать: `IngredientList` и `Instructions`. Теперь мы можем использовать их для создания компонента `Recipe`. Вместо длинного сложного кода, в котором весь рецепт был бы изложен в одном месте, мы получили декларативный код, состоящий их мелких компонентов. Такой код не только элегантен и прост, но и хорошо читается. Видно, что в рецепт входят название, список ингредиентов и инструкции по приготовлению. Подробности рендеринга ингредиентов и инструкций мы изложили в более мелких и простых компонентах.

При модульном подходе компонент `Menu` выглядит очень похоже, но находится в отдельном файле, может импортировать модули, которые ему нужны, и экспортировать себя:

```
// ./src/components/Menu.js

import React from "react";
import Recipe from "./Recipe";

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

```
    </article>
  );
}

export default Menu;
```

Мы по-прежнему используем ReactDOM для рендеринга компонента `Menu`. Главный файл проекта — index.js. Он будет отвечать за рендеринг компонента в DOM.

```
// ./src/index.js

import React from "react";
import { render } from "react-dom";
import Menu from "./components/Menu";
import data from "./data/recipes.json";

render(<Menu recipes={data} />, document.getElementById("root"));
```

Первые четыре оператора импортируют необходимые для работы приложения модули. Вместо того чтобы загружать `react` и `react-dom` через теги `script`, мы импортируем их, чтобы webpack мог добавить их в пакет. Также нам понадобится компонент `Menu` и образец массива данных, перенесенный в отдельный модуль. У нас все еще два рецепта: Baked Salmon и Fish Tacos.

Все наши импортированные переменные являются локальными для файла index.js. Когда мы отображаем компонент `Menu`, мы передаем массив данных рецепта этому компоненту как свойство.

Данные извлекаются из файла recipes.json. Это те же данные, которые мы использовали ранее в этой главе, но теперь они соответствуют действующим правилам форматирования JSON:

```
// ./src/data/recipes.json

[
  {
    "name": "Baked Salmon",
    "ingredients": [
      { "name": "Salmon", "amount": 1, "measurement": "lb" },
      { "name": "Pine Nuts", "amount": 1, "measurement": "cup" },
      { "name": "Butter Lettuce", "amount": 2, "measurement": "cups" },
      { "name": "Yellow Squash", "amount": 1, "measurement": "med" },
      { "name": "Olive Oil", "amount": 0.5, "measurement": "cup" },
      { "name": "Garlic", "amount": 3, "measurement": "cloves" }
    ],
    "steps": [
      "Preheat the oven to 350 degrees.",
      "Spread the olive oil around a glass baking dish.",
      "Add the yellow squash and place in the oven for 30 mins.",
```

```
      "Add the salmon, garlic, and pine nuts to the dish.",
      "Bake for 15 minutes.",
      "Remove from oven. Add the lettuce and serve."
    ]
  },
  {
    "name": "Fish Tacos",
    "ingredients": [
      { "name": "Whitefish", "amount": 1, "measurement": "lb" },
      { "name": "Cheese", "amount": 1, "measurement": "cup" },
      { "name": "Iceberg Lettuce", "amount": 2, "measurement": "cups" },
      { "name": "Tomatoes", "amount": 2, "measurement": "large" },
      { "name": "Tortillas", "amount": 3, "measurement": "med" }
    ],
    "steps": [
      "Cook the fish on the grill until cooked through.",
      "Place the fish on the 3 tortillas.",
      "Top them with lettuce, tomatoes, and cheese."
    ]
  }
]
```

Теперь, когда мы разделили код на отдельные модули и файлы, создадим процесс сборки с помощью webpack, который снова объединит все в один файл. Вы спросите: «То есть мы только что лезли из кожи вон, чтобы разбить все на части, а теперь собираемся использовать инструмент, чтобы собрать все вместе? Зачем?» Разделение проектов на отдельные файлы обычно упрощает управление более крупными проектами, поскольку члены команды могут работать над отдельными компонентами, не мешая друг другу. Кроме того, отдельные файлы легче тестировать.

3. Создание сборки webpack

Чтобы организовать статический процесс сборки с помощью webpack, установим несколько вещей. Все, что нам нужно, можно установить с помощью npm:

```
npm install --save-dev webpack webpack-cli
```

Помните, что мы уже установили React и ReactDOM.

Чтобы модульное приложение рецептов работало, дадим webpack указания о том, как объединить исходный код в один файл. Начиная с версии 4.0.0, объединение проекта в пакет webpack не требует включения файла конфигурации. webpack может взять для упаковки кода значения по умолчанию (webpack.config.js) или при включенном файле конфигурации позволить нам настроить этот процесс. Второй вариант покажет нам работу webpack.

Исходный файл приложения — index.js. В нем импортируются React, ReactDOM и файл Menu.js. Именно его мы запустим в браузере первым. Где бы webpack ни искал оператор импорта, он находит связанный модуль в файловой системе и включает его в пакет. В файле index.js импортируется Menu.js, в Menu.js импортируется Recipe.js, в Recipe.js импортируется Instructions.js и IngredientsList.js, а в IngredientsList.js импортируется Ingredient.js. Webpack прослеживает это дерево импорта и включает все необходимые модули в пакет. Обход всех этих файлов создает так называемый *граф зависимостей*. Зависимость — это то, что нужно приложению, например файл компонента, файл библиотеки вроде React или изображение. Если мы изобразим каждый файл, который нам нужен, в виде круга на графике, а затем webpack нарисует линии между кружками для создания графа, то это и будет пакет.

ОПЕРАТОРЫ IMPORT

Мы используем операторы `import`, которые в настоящее время не поддерживаются большинством браузеров и Node.js. Они работают лишь потому, что Babel преобразует их в операторы `require('module/path')`. Функция `require` обычно используется для загрузки модулей CommonJS.

Поскольку webpack собирает пакет, нужно указать ему преобразовать JSX в элементы React.

Файл webpack.config.js — это еще один модуль, который экспортирует литеральный объект JavaScript, описывающий действия, которые должен выполнять веб-пакет. Файл конфигурации нужно сохранить в корневой папке проекта, рядом с файлом index.js:

```
// ./webpack.config.js

var path = require("path");

module.exports = {
  entry: "./src/index.js",
  output: {
    path: path.join(__dirname, "dist", "assets"),
    filename: "bundle.js"
  }
};
```

Сначала мы сообщаем webpack, что наш входной файл клиента — ./src/index.js. webpack автоматически построит граф зависимостей на основе операторов `import`, начиная с этого файла. Затем мы указываем, что хотим вывести связанный файл JavaScript в ./dist/bundle.js. Именно сюда webpack поместит окончательный упакованный JavaScript.

Теперь установим необходимые зависимости для Babel. Нам понадобятся модули `babel-loader` и `@babel/core`:

```
npm install babel-loader @babel/core --save-dev
```

Следующий набор инструкций для webpack состоит из списка загрузчиков (loader), выполняющих запуск определенных модулей. Их мы добавим в файл конфигурации под полем `module`:

```
module.exports = {
  entry: "./src/index.js",
  output: {
    path: path.join(__dirname, "dist", "assets"),
    filename: "bundle.js"
  },
  module: {
    rules: [{ test: /\.js$/, exclude: /node_modules/, loader: "babel-loader" }]
  }
};
```

Поле `rules` представляет собой массив, позволяющий включить в webpack множество типов загрузчиков. В этом примере мы включаем только загрузчик `babel`. Каждый загрузчик представляет собой объект JavaScript. Поле `test` — это регулярное выражение, которое соответствует пути к файлу каждого модуля, с которым должен работать загрузчик. Мы запускаем `babel-loader` для всех импортированных файлов JavaScript, кроме тех, которые находятся в папке node_modules.

На этом этапе задаем предустановку для запуска Babel, тем самым сообщая Babel, какое преобразование нужно выполнить. Другими словами, мы говорим: «Привет, Babel. Если увидишь синтаксис ESNext, преобразуй его в синтаксис, который точно подойдет браузеру. Если найдешь JSX, транспилируй и его тоже». Начнем с предустановок:

```
npm install @babel/preset-env @babel/preset-react --save-dev
```

Затем создадим в корне проекта еще один файл: .babelrc:

```
{
  "presets": ["@babel/preset-env", "@babel/preset-react"]
}
```

Отлично! Мы создали проект, действительно напоминающий настоящее приложение React! Теперь запустим webpack, чтобы убедиться, что все работает.

Webpack запускается статически. Обычно пакеты создаются перед развертыванием приложения на сервере. Их можно запускать из командной строки, используя npx:

```
npx webpack --mode development
```

Webpack либо завершит работу успешно и создаст пакет, либо завершит работу с ошибкой и сообщит об этом. Большинство ошибок связано с неработающими ссылками на импорт. При отладке ошибок webpack проверьте имена файлов и пути к файлам, используемые в операторах `import`.

Вы также можете добавить сценарий npm в файл package.json для создания ярлыка:

```
"scripts": {
  "build": "webpack --mode production"
},
```

Затем вы можете запустить ярлык для создания пакета:

```
npm run build
```

Загрузка пакета

Пакет готов, и что дальше? Мы экспортировали пакет в папку dist. В ней содержатся файлы, которые мы хотим запустить на веб-сервере. В папке dist разместим файл index.html. Он должен включать целевой элемент `div`, в который будет монтироваться компонент React `Menu`. Также требуется один тег `script`, который загрузит наш связанный JavaScript:

```
// ./dist/index.html

<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8" />
    <title>React Recipes App</title>
  </head>
  <body>
    <div id="root"></div>
    <script src="bundle.js"></script>
  </body>
</html>
```

Это домашняя страница приложения. Она загрузит все необходимое из одного файла с помощью одного HTTP-запроса: bundle.js. Разверните эти файлы на своем веб-сервере или создайте приложение веб-сервера, которое будет обслуживать эти файлы с помощью чего-то вроде Node.js или Ruby on Rails.

Сопоставление источников

Объединение кода в один файл может вызвать проблемы отладки приложения в браузере. Устранить эту проблему поможет *карта исходников* (source map) — файл, который сопоставит пакет с оригинальными исходными файлами. В случае с webpack достаточно будет добавить пару строк в файл webpack.config.js.

```
//webpack.config.js with source mapping

module.exports = {
  ...
  devtool: "#source-map" // Add this option for source mapping
};
```

Установка значения `'#source-map'` для свойства `devtool` сообщит webpack, что вы хотите использовать сопоставление источников. В следующий раз, когда вы запустите webpack, то увидите два новых файла, которые сгенерированы и добавлены в папку dist: исходный файл bundle.js и bundle.js.map.

Карта исходников позволяет отлаживать исходные файлы. На вкладке Sources инструментов разработчика браузера есть папка webpack://. В ней вы найдете все исходные файлы вашего пакета (рис. 5.4).

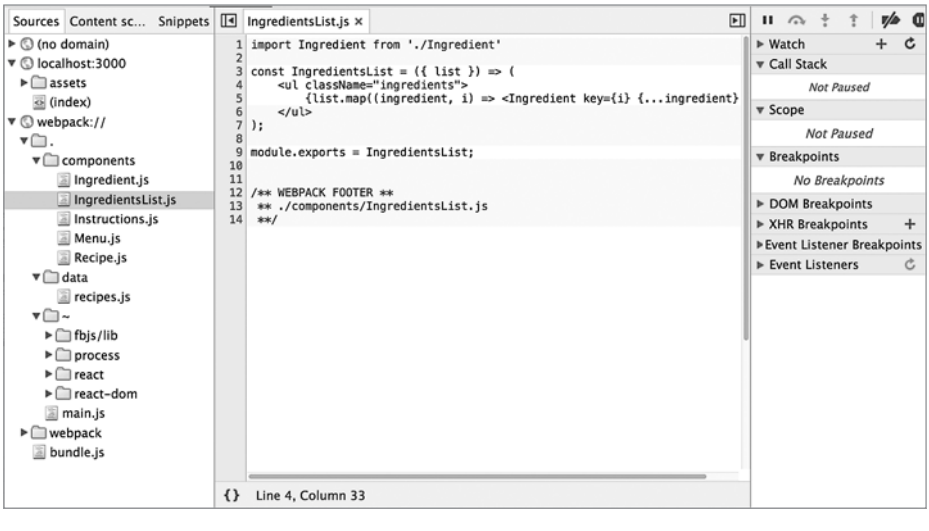


Рис. 5.4. Панель Sources в Chrome Developer Tools

Вы можете выполнять отладку из этих файлов с помощью встроенного в браузер пошагового отладчика. Щелкнув по любому номеру строки, вы можете добавить контрольную точку. При обновлении содержимого браузера, когда процесс выполнения кода дойдет до места установки любой контрольной точки в исходном файле, обработка кода JavaScript приостановится. Вы сможете проверить переменные в области видимости на панели Scope или добавить переменные для отслеживания на панели Watch.

Создание приложения React

Еще один удивительный и полезный для разработчика React инструмент — это Create React App. Это инструмент командной строки, который автоматически генерирует проект React. Create React App создан в стиле проекта Ember CLI (https://cli.emberjs.com/release/) и позволяет начать работу с проектом React быстро, без ручной настройки webpack, Babel, ESLint и похожих инструментов.

Чтобы начать работу с Create React App, установите пакет глобально:

```
npm install -g create-react-app
```

Затем введите команду и имя папки, в которой хотите создать приложение:

```
create-react-app my-project
```

NPX

Вы также можете использовать npx для запуска Create React App без необходимости глобальной установки пакета. Просто запустите команду `npx create-response-app my-project`.

Мы получим проект React в хранилище всего с тремя зависимостями: React, ReactDOM и `react-scripts`. Модуль `react-scripts` тоже создан Facebook, и именно он устанавливает Babel, ESLint, webpack и другие инструменты без необходимости их ручной настройки. В созданной папке проекта вы также найдете папку src, содержащую файл App.js. В нем вы можете редактировать корневой компонент и импортировать файлы других компонентов.

Из папки my-react-project запустите команду `npm start`. Если хотите, используйте `yarn start`, чтобы запустить приложение на порту 3000.

С помощью команд `npm test` или `yarn test` вы можете запустить тестовые файлы в интерактивном режиме.

Также вы можете запустить команду `npm run build`. Используя yarn, запустите `yarn build`.

В итоге будет создан готовый к продакшену пакет, предварительно преобразованный и минифицированный.

Create React App — отличный инструмент для начинающих и опытных разработчиков React. По мере развития инструмента, вероятно, в нем появится больше функций, поэтому следите за изменениями на GitHub (github.com/facebook/create-react-app). Еще один способ начать работу с React, не беспокоясь о настройке сборки веб-пакета, — использовать CodeSandbox — IDE, работающую в сети по адресу codesandbox.io.

В ходе изучения этой главы, посвященной JSX, вы улучшили свои навыки в React. Вы создали компоненты, разбили их на структуру проекта и узнали больше о Babel и webpack. Теперь вы готовы вывести свои знания о компонентах на новый уровень. Пора поговорить о хуках.

ГЛАВА 6

Управление состояниями

Данные оживляют компоненты React. Пользовательский интерфейс, который мы создали в предыдущей главе, будет бесполезен без самих рецептов, ингредиентов и инструкций по приготовлению блюд. Пользовательские интерфейсы — это инструменты, которые будут использовать создатели контента. Чтобы сделать их лучше, нам нужно знать, как эффективно манипулировать данными и изменять их.

В предыдущей главе мы построили *дерево компонентов* — иерархию компонентов, через которые данные могут проходить в виде свойств. Свойства — это половина картины. Другая половина — это состояния. *Состояние* приложения React определяется данными, способными изменяться. Введение состояний в приложение позволит пользователям создавать новые, изменять существующие и удалять старые рецепты.

Состояния и свойства тесно связаны. Работая с приложениями React, мы соединяем компоненты друг с другом на основе этой взаимосвязи. Когда состояние дерева компонентов изменяется, меняются и его свойства. Новые данные проходят через дерево, вызывая рендеринг определенных листьев и ветвей с новым контентом.

В этой главе мы будем оживлять приложения, вводя в них состояние. Научимся создавать компоненты с отслеживанием состояния и узнаем, как состояние может быть передано вниз по дереву компонентов, а действия пользователя — вверх. Изучим методы сбора данных с форм от пользователей. Рассмотрим способы разделения задач в приложениях, введя поставщиков контекста с отслеживанием состояния.

Создание компонента системы оценок

Не будь системы оценок, мы ели бы отвратительную еду и смотрели бы ужасные фильмы. Если мы хотим позволить пользователям управлять контентом на на-

шем веб-сайте, нужен способ узнать, является ли данный контент хорошим или нет. Компонент `StarRating` будет одним из самых важных компонентов React, которые вы когда-либо создавали (рис. 6.1).

Рис. 6.1. Компонент StarRating

Компонент `StarRating` позволяет пользователям выставлять контенту оценку, нажимая на звездочки. Плохой контент получает одну звезду. Хороший и рекомендуемый контент получает пять звезд. Итак, нам понадобится звездочка, которую мы возьмем из react-icons:

```
npm i react-icons
```

react-icons — это библиотека npm, содержащая сотни SVG-иконок, которые распределяются как компоненты React. Устанавливая ее, мы устанавливаем несколько популярных библиотек SVG-иконок (https://react-icons.github.io/react-icons/). Выберем звездочку из коллекции Font Awesome:

```
import React from "react";
import { FaStar } from "react-icons/fa";

export default function StarRating() {
  return [
    <FaStar color="red" />,
    <FaStar color="red" />,
    <FaStar color="red" />,
    <FaStar color="grey" />,
    <FaStar color="grey" />
  ];
}
```

Мы создали компонент `StarRating`, который отображает пять звезд SVG, импортированных из react-icons. Первые три звезды закрашены красным цветом, а последние две — серым. Сначала мы отображаем звезды, потому что их вид дает понять, что строить дальше. Выделенная пользователем звезда должна быть закрашена красным, а невыделенная — серым. Создадим компонент, который автоматически сохраняет звезды в зависимости от выбранного свойства:

```
const Star = ({ selected = false }) => (
  <FaStar color={selected ? "red" : "grey"} />
);
```

Компонент `Star` отображает отдельную звезду и использует свойство `selected` для ее заливки соответствующим цветом. Если выбранное свойство не передается в этот компонент, мы предполагаем, что звездочка не выделена и по умолчанию будет заполнена серым цветом.

Пятизвездочная рейтинговая система довольно популярна, но десятизвездочная система — более подробна. Позволим разработчикам выбирать количество звездочек. Добавим свойство `totalStars` в компонент `StarRating`:

```
const createArray = length => [...Array(length)];

export default function StarRating({ totalStars = 5 }) {
  return createArray(totalStars).map((n, i) => <Star key={i} />);
}
```

Мы добавили функцию `createArray` из главы 2. Указав длину массива, который мы хотим создать, в свойстве `totalStars` этой функции мы получим новый массив нужной длины. Отобразим этот массив и компоненты `Star`. По умолчанию `totalStars` имеет значение `5`, значит, будет отображено пять серых звезд (рис. 6.2).

Рис. 6.2. Пять звезд

Хук useState

Пришло время сделать компонент `StarRating` интерактивным, чтобы позволить пользователям ставить оценку. Поскольку оценка — это значение, которое будет изменяться, мы будем сохранять и изменять его, используя состояние React. Включим состояние в функциональный компонент, используя *хук* (hook). Хуки содержат логику повторно используемого кода, отдельную от дерева компонентов. Они позволяют добавлять компонентам функциональность. В React есть несколько встроенных хуков, которые можно использовать прямо из коробки. В данном случае мы хотим добавить компоненту состояние, поэтому первый хук, с которым мы будем работать, это `useState`. Этот хук уже доступен в пакете `react`, и нам просто нужно импортировать его:

```
import React, { useState } from "react";
import { FaStar } from "react-icons/fa";
```

Звезды, на которые нажал пользователь, являются оценкой. Мы создадим переменную состояния `selectedStars`, в которой будет храниться оценка пользователя, и добавим хук `useState` непосредственно в компонент `StarRating`:

```
export default function StarRating({ totalStars = 5 }) {
  const [selectedStars] = useState(3);
  return (
    <>
      {createArray(totalStars).map((n, i) => (
        <Star key={i} selected={selectedStars > i} />
      ))}
      <p>
        {selectedStars} of {totalStars} stars
      </p>
    </>
  );
}
```

Только что мы *хукнули* (hooked), то есть подключили, компонент к состоянию. Хук `useState` — это функция, которую мы можем вызвать для возврата массива. Первое значение этого массива — переменная состояния, которую мы хотим использовать, `selectedStars`, то есть количество закрашенных красным звездочек в оценке `StarRating`. `useState` возвращает массив. Деструктуризация массива позволит называть переменную состояния как угодно. Значение, которое мы отправляем функции `useState`, является значением по умолчанию для переменной состояния. В этом случае `selectedStars` будет изначально равно `3` (рис. 6.3).

Рис. 6.3. Выбраны три звезды из пяти

Чтобы получить другую оценку, разрешим пользователю нажимать на любую из звездочек. Сделаем звезды интерактивными, добавив обработчик `onClick` в компонент `FaStar`:

```
const Star = ({ selected = false, onSelect = f => f }) => (
  <FaStar color={selected ? "red" : "grey"} onClick={onSelect} />
);
```

Мы добавили звездам свойство `onSelect`. Проверьте: это свойство является функцией. Когда пользователь щелкает по компоненту `FaStar`, мы вызываем эту функцию, которая уведомляет своего родителя о том, что была нажата звездочка. Значение функции по умолчанию `f => f`. Это просто функция-пустышка, которая ничего не делает, а только возвращает любой аргумент, отправленный ей. Однако, если мы не установим функцию по умолчанию и свойство `onSelect` не будет определено, при щелчке по компоненту `FaStar` произойдет ошибка, поскольку значение `onSelect` должно быть функцией. Несмотря на то что `f => f` ничего не делает, это все же функция, которую можно вызывать без ошибок. Если свойство `onSelect` не определено, проблем не будет. React просто вызовет фальшивую функцию, и ничего не произойдет.

Теперь, когда на компонент `Star` можно нажимать, мы будем использовать его, чтобы изменить состояние `StarRating`:

```
export default function StarRating({ totalStars = 5 }) {
  const [selectedStars, setSelectedStars] = useState(0);
  return (
    <>
      {createArray(totalStars).map((n, i) => (
        <Star
          key={i}
          selected={selectedStars > i}
          onSelect={() => setSelectedStars(i + 1)}
        />
      ))}
      <p>
        {selectedStars} of {totalStars} stars
      </p>
    </>
  );
}
```

Нам понадобится функция, которая может изменять значение `selectedStars`. Второй элемент массива, возвращаемый хуком `useState`, — это функция, которую можно использовать для изменения значения состояния. Опять же, деструктурируя этот массив, мы сможем назвать эту функцию как угодно. В данном случае мы называем ее `setSelectedStars`, потому что она устанавливает значение `selectedStars`.

Самое важное — помнить, что хуки могут вызывать повторную визуализацию компонента, к которому они подключены. Каждый раз, когда мы вызываем функцию `setSelectedStars` для изменения значения `selectedStars`, компонент функции `StarRating` будет повторно активирован хуком и начнет заново отображаться с новым значением `selectedStars`. Вот почему хуки — такая крутая

штука. Когда данные в хуке меняются, хук может повторно отобразить связанный компонент с новыми данными.

Компонент `StarRating` будет повторно отображаться каждый раз, когда пользователь нажмет звездочку, поскольку при этом действии вызывается свойство `onSelect` звездочки, которое, в свою очередь, вызывает функцию `setSelectedStars` и отправляет ей номер только что выбранной звезды. Мы можем использовать переменную `i` из функции `map`, чтобы вычислить количество звезд. Когда функция `map` отображает первую звезду, значение `i` равно `0`. Это означает, что нам нужно добавить к этому значению `1`, чтобы получить правильное количество звезд. Когда вызывается `setSelectedStars`, вызывается компонент `StarRating` со значением `selectedStars` (рис. 6.4).

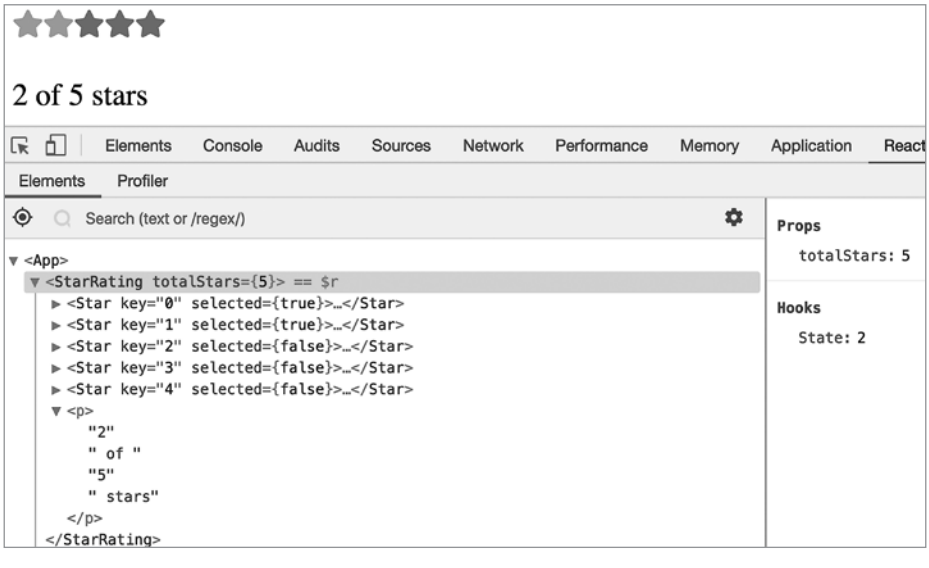


Рис. 6.4. Хуки в инструментах разработчика React

Инструменты разработчика React позволяют увидеть, какие хуки включены в определенные компоненты. Отображая компонент `StarRating` в браузере, мы можем просматривать отладочную информацию о компоненте, выбирая его в инструментах разработчика. В столбце справа мы видим, что компонент `StarRating` включает в себя обработчик состояния, имеющий значение `2`. Во время взаимодействия с приложением мы можем наблюдать за изменением значения состояния и повторной визуализацией дерева компонентов с соответствующим количеством выбранных звезд.

СОСТОЯНИЯ В REACT РАНЬШЕ

В старых версиях React до 16.8.0 добавить состояние к компоненту можно было только с помощью компонента класса. Для этого требовалось много кода, и такой путь усложнял повторное использование функций в компонентах. Хуки были разработаны для решения проблем, связанных с компонентами класса, так как они позволили добавить функциональность в функциональные компоненты.

Приведенный ниже код — это компонент класса. Это оригинальный компонент `StarRating`, который мы печатали в первом издании этой книги:

```
import React, { Component } from "react";

export default class StarRating extends Component {
  constructor(props) {
    super(props);
    this.state = {
      starsSelected: 0
    };
    this.change = this.change.bind(this);
  }

  change(starsSelected) {
    this.setState({ starsSelected });
  }

  render() {
    const { totalStars } = this.props;
    const { starsSelected } = this.state;
    return (
      <div>
        {[...Array(totalStars)].map((n, i) => (
          <Star
            key={i}
            selected={i < starsSelected}
            onClick={() => this.change(i + 1)}
          />
        ))}
        <p>
          {starsSelected} of {totalStars} stars
        </p>
      </div>
    );
  }
}
```

Этот компонент класса делает то же самое, что и наш функциональный компонент, но для него требуется гораздо больше кода. Кроме того, он содержит путаницу из-за использования привязки функции и ключевого слова `this`.

На сегодняшний день этот код все еще работает. Но в этой книге мы не рассматриваем компоненты класса, так как они больше не нужны. Функциональные компоненты и хуки — это будущее React, и мы не оглядываемся назад. Может наступить день, когда компоненты класса будут официально объявлены устаревшими, и этот код перестанет поддерживаться.

Рефакторинг для улучшения повторного использования

Прямо сейчас компонент `Star` готов к работе. Вы можете использовать его в нескольких приложениях, в которых нужны оценки от пользователей. Однако если бы мы отправили этот компонент в npm, любой человек в мире смог бы применить его для получения пользовательских оценок, поэтому рассмотрим еще парочку вариантов, когда он может пригодиться.

Сначала рассмотрим свойство `style`, которое позволяет добавлять к элементам стили CSS. Вполне возможно, что будущий разработчик (или вы сами) захочет изменить стиль контейнера примерно так:

```
export default function App() {
  return <StarRating style={{ backgroundColor: "lightblue" }} />;
}
```

Все элементы React и многие компоненты имеют свойства, отвечающие за стиль. Поэтому попытка изменить стиль общего компонента кажется разумной.

Нам нужно лишь собрать эти стили и передать их в `StarRating`. Сейчас у `StarRating` нет единого контейнера, потому что мы используем фрагмент React. Чтобы передать ему стили, нам нужно перейти от фрагментов к элементу `div` и передать ему:

```
export default function StarRating({ style = {}, totalStars = 5 }) {
  const [selectedStars, setSelectedStars] = useState(0);
  return (
    <div style={{ padding: "5px", ...style }}>
      {createArray(totalStars).map((n, i) => (
        <Star
          key={i}
          selected={selectedStars > i}
          onSelect={() => setSelectedStars(i + 1)}
        />
      ))}
      <p>
        {selectedStars} of {totalStars} stars
      </p>
    </div>
  );
}
```

Мы заменили фрагмент элементом `div`, а затем применили к нему стили. По умолчанию мы назначили `div` отступ в 5 пикселов, а затем с помощью оператора распространения применили остальные свойства объекта `style` к стилю `div`.

Некоторые разработчики попытаются реализовать другие общие свойства для компонента:

```
export default function App() {
  return (
    <StarRating
      style={{ backgroundColor: "lightblue" }}
      onDoubleClick={e => alert("double click")}
    />
  );
}
```

В этом примере пользователь пытается добавить метод двойного щелчка ко всему компоненту `StarRating`. Если это необходимо, мы можем передать этот метод и его свойства в родительский `div`:

```
export default function StarRating({ style = {}, totalStars = 5, ...props }) {
  const [selectedStars, setSelectedStars] = useState(0);
  return (
    <div style={{ padding: 5, ...style }} {...props}>
      ...
    </div>
  );
}
```

Сначала мы соберем все свойства, которые пользователь попытается добавить в `StarRating`, с помощью оператора распространения: `... props`, а затем передадим их элементу `div` — `{... props}`.

Здесь мы сделаем два предположения: что пользователь добавит только те свойства, которые поддерживаются элементом `div`, и что он не станет или не сможет добавлять вредоносные свойства.

Это не правило для работы с компонентами. Добавление такого уровня поддержки — хорошая идея, но нужно учитывать возможные варианты использования компонента в будущем.

Состояние в деревьях компонентов

Использование состояний в отдельных компонентах — не лучшая идея. Распределение данных о состоянии по слишком большому количеству компонентов затруднит поиск ошибок и внесение изменений в приложение. Дело в том, что отслеживать, где находятся значения состояния в дереве компонентов, затруднительно. Легче оценить состояние приложения или конкретной функции. У этой методологии есть несколько подходов, и первый, который мы проана-

лизируем, — сохранение состояния в корне дерева компонентов и передача его дочерним компонентам через свойства.

Создадим небольшое приложение для сохранения списка цветов «Color Organizer», которое позволит пользователям связывать список цветов с настраиваемым заголовком и оценкой. Для начала образец набора данных может выглядеть так:

```
[
  {
    "id": "0175d1f0-a8c6-41bf-8d02-df5734d829a4",
    "title": "ocean at dusk",
    "color": "#00c4e2",
    "rating": 5
  },
  {
    "id": "83c7ba2f-7392-4d7d-9e23-35adbe186046",
    "title": "lawn",
    "color": "#26ac56",
    "rating": 3
  },
  {
    "id": "a11e3995-b0bd-4d58-8c48-5e49ae7f7f23",
    "title": "bright red",
    "color": "#ff0000",
    "rating": 0
  }
]
```

Файл color-data.json содержит массив из трех цветов. У каждого цвета есть атрибуты `id`, `title`, `color` и `rating`. Сначала мы создадим пользовательский интерфейс из компонентов React, которые будут использоваться для рендеринга этих данных в браузере. Затем мы позволим пользователям добавлять новые цвета, а также оценивать и удалять цвета из списка.

Передача состояния вниз по дереву компонентов

В этой итерации мы сохраним состояние в корне Color Organizer, в компоненте `App`, и передадим дочерним компонентам цвета для рендеринга. Компонент `App` будет единственным в приложении компонентом, хранящим состояние. Добавим список цветов в `App` с помощью хука `useState`:

```
import React, { useState } from "react";
import colorData from "./color-data.json";
import ColorList from "./ColorList.js";

export default function App() {
```

```
  const [colors] = useState(colorData);
  return <ColorList colors={colors} />;
}
```

Компонент App находится в корне дерева. Добавление useState подключает его к управлению состоянием цветов. colorData — это массив цветов, который App использует как начальное состояние для свойства colors. Затем цвета передаются в компонент ColorList:

```
import React from "react";
import Color from "./Color";

export default function ColorList({ colors = [] }) {
  if(!colors.length) return <div>No Colors Listed.</div>;
  return (
    <div>
      {
        colors.map(color => <Color key={color.id} {...color} />)
      }
    </div>
  );
}
```

ColorList получает цвета от App в качестве свойств. Если список пуст, он отобразит сообщение для пользователей. Имея массив цветов, мы можем отобразить его и передать детали каждого цвета дальше по дереву в компонент Color:

```
export default function Color({ title, color, rating }) {
  return (
    <section>
      <h1>{title}</h1>
      <div style={{ height: 50, backgroundColor: color }} />
      <StarRating selectedStars={rating} />
    </section>
  );
}
```

Компонент Color ожидает три свойства: title, color и rating. Эти значения находятся в объекте каждого цвета и передаются компоненту с помощью оператора распространения <Color {... color} />. Он берет поля из объекта color и передает их компоненту Color в виде свойств с тем же именем, что и ключ объекта. Компонент Color отображает эти значения. Свойство title отображается внутри элемента h1. Значение color отображается как backgroundColor элемента div. Свойство rating передается дальше вниз по дереву компоненту StarRating, который отображает оценку в виде выбранных звезд:

```
export default function StarRating({ totalStars = 5, selectedStars = 0 }) {
  return (
    <>
```

```
      {createArray(totalStars).map((n, i) => (
        <Star
          key={i}
          selected={selectedStars > i}
        />
      ))}
      <p>
        {selectedStars} of {totalStars} stars
      </p>
    </>
  );
}
```

Компонент `StarRating` был изменен. Мы превратили его в чистый компонент — функциональный компонент, который не содержит состояния и будет при одних и тех же заданных свойствах отображать один и тот же пользовательский интерфейс. Мы сделали это именно так, потому что состояние цвета оценки хранится в массиве `colors` в корне дерева компонентов. Помните, что наша цель — сохранить состояние в одном месте, а не распределить его по множеству компонентов в дереве.

Компонент `StarRating` может сохранять собственное состояние и получать состояние от родительского компонента через свойства. Обычно это необходимо при распространении компонентов для более широкого использования сообществом. Мы продемонстрируем эту технику в следующей главе, когда рассмотрим хук `useEffecthook`.

На этом этапе мы закончили передачу состояния вниз по дереву компонентов от компонента `App` до каждого компонента `Star`, который закрашен красным, чтобы визуально представить рейтинг для каждого цвета. Если мы визуализируем приложение на основе файла color-data.json, который был указан ранее, мы увидим наши цвета в браузере (рис. 6.5).

Передача взаимодействий вверх по дереву компонентов

Мы визуализировали представление массива `colors` в виде пользовательского интерфейса, используя компоненты React и передав данные вниз по дереву от родительского компонента к дочернему через свойства. Что произойдет, если мы захотим удалить цвет из списка или изменить его оценку? Массив `colors` хранится в состоянии в корне дерева. Нам нужно собрать все взаимодействия от дочерних компонентов и передать их обратно по дереву в корневой компонент, где мы сможем изменить состояние.

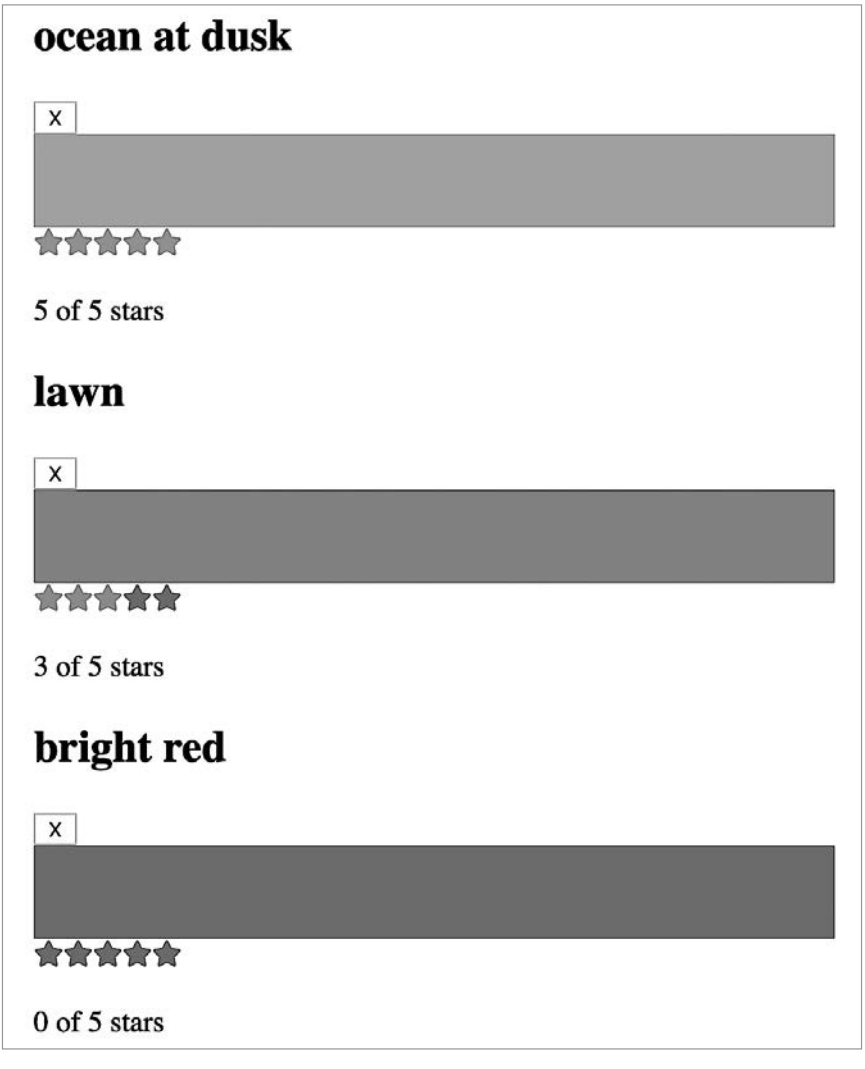


Рис. 6.5. Рендеринг Color Organizer в браузере

Например, предположим, что в компонент `Color` мы добавили кнопку `Remove` рядом с заголовком каждого цвета, чтобы пользователи могли удалять цвета из состояния:

```
import { FaTrash } from "react-icons/fa";

export default function Color({ id, title, color, rating, onRemove = f => f }) {
  return (
```

```
    <section>
      <h1>{title}</h1>
      <button onClick={() => onRemove(id)}>
        <FaTrash />
      </button>
      <div style={{ height: 50, backgroundColor: color }} />
      <StarRating selectedStars={rating} />
    </section>
  );
}
```

Мы изменили цвет, добавив кнопку, которая позволяет пользователям удалять цвета. Сначала мы импортировали значок корзины из react-icons. Затем обернули значок `FaTrash` в кнопку. Добавление обработчика `onClick` к этой кнопке позволяет вызывать свойство функции `onRemove`, которое было добавлено в список свойств вместе с идентификатором. Когда пользователь нажимает кнопку Remove, мы вызываем функцию `removeColor` и передаем ей идентификатор цвета, который хотим удалить. Для этого значение `id` также было получено из свойств компонента `Color`.

Это отличное решение, потому что компонент сохраняет чистоту `Color`. У него нет состояния, и его легко снова использовать в другой части приложения или в другом приложении. Компонент `Color` не знает, что происходит, когда пользователь нажимает кнопку `Remove`. Он лишь уведомляет родителя о том, что это событие произошло, и сообщает, какой цвет пользователь хочет удалить. Дальше ответственность за обработку этого события лежит на родительском компоненте:

```
export default function ColorList({ colors = [], onRemoveColor = f => f }) {
  if (!colors.length) return <div>No Colors Listed. (Add a Color)</div>;

return (
    colors.map(color => (
          <Color key={color.id} {...color} onRemove={onRemoveColor} />
        )
      }
    </div>
  );
}
```

Родителем компонента `Color` является `ColorList`. Он тоже не имеет доступа к состоянию. Вместо того чтобы удалять цвет, он просто передает событие своему родителю, благодаря добавлению свойства функции `onRemoveColor`. Если компонент `Color` вызывает свойство `onRemove`, `ColorList`, в свою очередь, вызывает свое событие `onRemoveColor` и передает ответственность за удаление цвета своему родителю. Идентификатор цвета все еще передается в функцию `onRemoveColor`.

Родителем `ColorList` является `App`. Этот компонент уже связан с состоянием. Здесь мы можем взять идентификатор цвета и удалить цвет в состоянии:

```
export default function App() {
  const [colors, setColors] = useState(colorData);
  return (
    <ColorList
      colors={colors}
      onRemoveColor={id => {
        const newColors = colors.filter(color => color.id !== id);
        setColors(newColors);
      }}
    />
  );
}
```

Сначала добавим переменную `setColors`. Помните, что второй аргумент в массиве, возвращаемом `useState`, — это функция, которую мы можем использовать для изменения состояния. Когда `ColorList` вызывает событие `onRemoveColor`, мы фиксируем идентификатор цвета, который нужно удалить из аргументов, и используем его для фильтрации списка цветов, исключая цвет, который пользователь хочет удалить. Далее меняем состояние. Мы используем функцию `setColors`, чтобы заменить массив цветов новым, отфильтрованным.

Изменение состояния массива `colors` вызывает повторный рендеринг компонента `App` с новым списком цветов. Новые цвета передаются компоненту `ColorList`, который также отображается заново. Он будет отображать компоненты `Color` оставшихся цветов, а в интерфейсе появятся внесенные нами изменения (будет отображено на один цвет меньше).

Если мы хотим оценить цвета `colors`, которые хранятся в состоянии компонентов `App`, нам придется повторить процесс с событием `onRate`. Сначала мы возьмем оценку от звезды, на которую щелкнули, и передадим это значение родительскому элементу `StarRating`:

```
export default function StarRating({
  totalStars = 5,
  selectedStars = 0,
  onRate = f => f
}) {
  return (
    <>
      {createArray(totalStars).map((n, i) => (
        <Star
          key={i}
          selected={selectedStars > i}
          onSelect={() => onRate(i + 1)}
        />
```

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

Затем мы возьмем оценку из обработчика `onRate`, который мы добавили в компонент `StarRating`. После этого мы передадим новую оценку вместе с `id` оцениваемого цвета до уровня родительского компонента `Color`, через другое свойство функции `onRate`:

```
export default function Color({
  id,
  title,
  color,
  rating,
  onRemove = f => f,
  onRate = f => f
}) {
  return (
    <section>
      <h1>{title}</h1>
      <button onClick={() => onRemove(id)}>
        <FaTrash />
      </button>
      <div style={{ height: 50, backgroundColor: color }} />
      <StarRating
        selectedStars={rating}
        onRate={rating => onRate(id, rating)}
      />
    </section>
  );
}
```

В компоненте `ColorList` нам нужно будет выхватить событие `onRate` из отдельных цветовых компонентов и передать их его родительскому элементу через свойство функции `onRateColor`:

```
export default function ColorList({
  colors = [],
  onRemoveColor = f => f,
  onRateColor = f => f
}) {
if (!colors.length) return <div>No Colors Listed. (Add a Color)</div>;
  return (
    <div className="color-list">
      {
        colors.map(color => (
          <Color
            key={color.id}
            {...color}
            onRemove={onRemoveColor}
```

```
            onRate={onRateColor}
          />
        )
      }
    </div>
  );
}
```

Наконец, пропустив событие через все эти компоненты, мы доберемся до компонента `App`, в котором сохраняется состояние и может быть сохранена новая оценка:

```
export default function App() {
  const [colors, setColors] = useState(colorData);
  return (
    <ColorList
      colors={colors}
      onRateColor={(id, rating) => {
        const newColors = colors.map(color =>
          color.id === id ? { ...color, rating } : color
        );
        setColors(newColors);
      }}
      onRemoveColor={id => {
        const newColors = colors.filter(color => color.id !== id);
        setColors(newColors);
      }}
    />
  );
}
```

Компонент `App` изменит оценки цветов, когда `ColorList` вызовет свойство `onRateColor` с `id` цвета и новой оценкой. Мы будем использовать эти значения для создания массива новых цветов путем сопоставления существующих цветов и изменения оценки цвета, соответствующего свойству `id`. Как только мы отправим `newColors` в функцию `setColors`, значение состояния цветов изменится, и компонент `App` будет вызываться с новым массивом `colors`.

Когда состояние массива `colors` изменится, дерево пользовательского интерфейса отобразится уже с новыми данными. Новая оценка будет отображена в виде красных звездочек. Так же, как мы передавали данные вниз по дереву компонентов через свойства, взаимодействия могут передаваться обратно по дереву вместе с данными через свойства функции.

Создание форм

Для многих из нас быть веб-разработчиком означает собирать большие объемы информации от пользователей с помощью форм. Если вы занимаетесь именно

этим, то вам придется создавать множество компонентов формы с помощью React. Все элементы формы HTML, доступные для DOM также доступны в виде элементов React, и, возможно, вы уже представляете, как визуализировать форму с помощью JSX:

```
<form>
  <input type="text" placeholder="color title..." required />
  <input type="color" required />
  <button>ADD</button>
</form>
```

Элемент `form` имеет три дочерних элемента: два элемента `input` и элемент `button`. Первый `input` — это текстовое поле для заголовков новых цветов. Второй `input` — это HTML-форма, которая позволяет пользователям выбирать цвет из цветового круга. Мы будем использовать базовую проверку HTML-формы, поэтому мы пометили оба `input` как `required`. Кнопка `ADD` будет использоваться для добавления нового цвета.

Использование ссылок

Для создания компонента формы в React можно применить несколько паттернов. Один из них включает доступ к узлу DOM напрямую с помощью функции React, называемой `refs`. В React ссылка — это объект, который хранит значения в течение всего времени жизни компонента. Есть несколько вариантов использования, в которых применяются ссылки. В этом разделе мы рассмотрим, как получить доступ к узлу DOM напрямую с помощью ссылки.

React предоставляет нам хук `useRef`, который мы можем использовать для создания *ссылки*. Мы используем этот хук при создании компонента `AddColorForm`:

```
import React, { useRef } from "react";

export default function AddColorForm({ onNewColor = f => f }) {
  const txtTitle = useRef();
  const hexColor = useRef();

  const submit = e => { ... }

  return (...)
}
```

При создании этого компонента мы также создаем две ссылки с помощью хука `useRef`: `txtTitle` ссылается на текстовое поле, которое мы добавили в форму для ввода заголовка, а `hexColor` обеспечивает доступ к шестнадцатеричным значениям цвета HTML. Мы можем установить значения для этих ссылок прямо в JSX, используя свойство `ref`:

```
return (
  <form onSubmit={submit}>
    <input ref={txtTitle} type="text" placeholder="color title..." required />
    <input ref={hexColor} type="color" required />
    <button>ADD</button>
  </form>
  );
}
```

Здесь мы устанавливаем значение для ссылок `txtTitle` и `hexColor`, добавляя атрибут `ref` к этим входным элементам в JSX. В результате будет создано поле `current` в объекте `ref`, которое будет напрямую ссылаться на элемент DOM. Это даст нам доступ к элементу DOM, а значит, мы сможем фиксировать его значение. Когда пользователь отправит эту форму, нажав кнопку `ADD`, мы вызовем функцию `submit`:

```
const submit = e => {
  e.preventDefault();
  const title = txtTitle.current.value;
  const color = hexColor.current.value;
  onNewColor(title, color);
  txtTitle.current.value = "";
  hexColor.current.value = "";
};
```

Когда мы отправляем формы HTML, по умолчанию они делают POST-запрос на текущий URL-адрес со значениями элементов формы, хранящимися в контейнере `body`. Нам это не нужно. Поэтому первой строкой кода в функции отправки мы сделаем `e.preventDefault()`. Она не даст браузеру отправить форму с помощью POST-запроса.

Затем мы фиксируем текущие значения для каждого элемента формы, используя их ссылки. Эти значения передаются родительскому элементу компонента через свойство функции `onNewColor`. И заголовок, и шестнадцатеричное значение нового цвета передаются как аргументы функции. Наконец, мы сбрасываем атрибут `value` для обоих компонентов `input`, чтобы очистить форму под новый ввод.

Вы заметили сдвиг парадигмы, который произошел при использовании ссылок? Мы изменили атрибут `value` узлов DOM напрямую, установив пустые строки. Это императивный код. Теперь компонент `AddColorForm` является *неконтролируемым компонентом*, потому что он использует DOM для сохранения значений формы. Иногда использование неконтролируемого компонента может избавить от проблем. Например, вы можете предоставить общий доступ к форме и ее значениям какому-то коду за пределами React. Но все равно в общем случае лучше использовать управляемые компоненты.

Контролируемые компоненты

В *контролируемом компоненте* значения форм управляются React, а не DOM. При этом не требуется использование ссылок и написание императивного кода. При работе с управляемым компонентом намного проще добавлять дополнительные функции, например надежную проверку формы. Изменим компонент `AddColorForm`, предоставив ему контроль над состоянием формы:

```
import React, { useState } from "react";
export default function AddColorForm({ onNewColor = f => f }) {
  const [title, setTitle] = useState("");
  const [color, setColor] = useState("#000000");
  const submit = e => { ... };
  return ( ... );
}
```

Вместо использования ссылок мы сохраним значения `title` и `color`, используя состояние React. Создадим переменные `title` и `color` и определим функции для изменения состояния: `setTitle` и `setColor`.

Теперь, когда компонент управляет значениями `title` и `color`, мы можем отображать их внутри элементов ввода, установив атрибут `value`. После того как мы установим атрибут `value` элемента `input`, мы больше не сможем изменять форму. Единственный способ изменить значение на этом этапе — изменять переменную состояния при каждом пользовательском вводе нового символа в элемент:

```
<form onSubmit={submit}>
  <input
    value={title}
    onChange={event => setTitle(event.target.value)}
    type="text"
    placeholder="color title..."
    required
  />
  <input
    value={color}
    onChange={event => setColor(event.target.value)}
    type="color"
    required
  />
  <button>ADD</button>
</form>
}
```

Этот управляемый компонент теперь устанавливает значение обоих элементов `input`, используя значения `title` и `color` из состояния. Всякий раз, когда эти элементы будут вызывать событие `onChange`, мы сможем получить доступ к новому значению с помощью аргумента `event`. `Event.target` — это ссылка на элемент DOM, и мы можем получить текущее значение этого элемента с по-

мощью свойства `event.target.value`. Когда заголовок изменится, мы вызовем `setTitle`, чтобы изменить значение заголовка в состоянии. Изменение этого значения инициирует повторную отрисовку компонента, и мы сможем отобразить новое значение для `title` внутри элемента `input`. Изменение цвета происходит точно так же.

Когда придет время отправить форму, мы сможем просто передать значения состояния для `title` и `color` в свойство функции `onNewColor` в качестве аргументов при ее вызове. Функции `set Title` и `setColor` могут использоваться для сброса значений после передачи цвета родительскому компоненту:

```
const submit = e => {
  e.preventDefault();
  onNewColor(title, color);
  setTitle("");
  setColor("");
};
```

Такой компонент называется управляемым компонентом, потому что React контролирует состояние формы. Стоит отметить, что контролируемые компоненты формы часто отображаются заново. Подумайте об этом: каждый новый символ, вводимый в поле заголовка, вызывает повторный рендеринг `AddColorForm`. Использование цветового круга приводит к тому, что этот компонент меняет значение чаще, чем `title`, потому что значение цвета постоянно изменяется, когда пользователь перемещает мышь по цветовому кругу. Это нормально — React предназначен для обработки такой нагрузки. Надеюсь, знание того, что управляемые компоненты часто отображаются заново, позволит вам не вешать на эти компоненты слишком тяжелые процессы. По крайней мере, эти знания пригодятся, когда вы будете пытаться оптимизировать компоненты React.

Создание собственных хуков

Когда у вас есть большая форма с большим количеством элементов `input`, может возникнуть соблазн скопировать и вставить эти две строки кода:

```
value={title}
onChange={event => setTitle(event.target.value)}
```

Может показаться, что работа идет быстрее, если просто копировать и вставлять эти свойства в каждый элемент формы, меняя лишь имена переменных. Однако всякий раз, когда вы копируете и вставляете код, вы должны воспринимать это как сигнал тревоги: в коде есть избыточность, достаточная, чтобы абстрагировать ее в функции.

Мы можем упаковать процессы, необходимые для создания компонентов контролируемой формы, в специальный хук `useInput`:

```
import { useState } from "react";

export const useInput = initialValue => {
  const [value, setValue] = useState(initialValue);
  return [
    { value, onChange: e => setValue(e.target.value) },
    () => setValue(initialValue)
  ];
};
```

Это пользовательский хук. Для его создания не требуется много кода. Внутри хука мы все еще используем хук `useState` для создания `value` состояния. Затем мы возвращаем массив. Первое значение массива — это объект, содержащий те же свойства, которые мы пытались скопировать и вставить: `value` из состояния вместе со свойством функции `onChange`, которая изменяет это значение в состоянии. Второе значение в массиве — это функция, которую можно использовать для установки значения по умолчанию. Мы можем использовать наш хук внутри `AddColorForm`:

```
import React from "react";
import { useInput } from "./hooks";

export default function AddColorForm({ onNewColor = f => f }) {
  const [titleProps, resetTitle] = useInput("");
  const [colorProps, resetColor] = useInput("#000000");
  const submit = event => { ... }
  return ( ... )
}
```

Хук `useState` инкапсулирован внутри хука `useInput`. Мы можем получить свойства для заголовка и цвета, деструктурируя их из первого значения возвращенного массива. Второе значение этого массива содержит функцию для сброса свойства `value` обратно к его начальному значению — пустой строке. `TitleProps` и `colorProps` готовы к распространению в соответствующие им входные элементы:

```
return (
  <form onSubmit={submit}>
    <input
      {...titleProps}
      type="text"
      placeholder="color title..."
      required
    />
    <input {...colorProps} type="color" required />
    <button>ADD</button>
  </form>
);
}
```

Распространение этих свойств с помощью настраиваемого хука намного интереснее, чем их копирование. Теперь в оба элемента ввода поступают значения для свойств и событий `onChange`. Мы использовали хук для создания управляемых элементов ввода, не беспокоясь о деталях базовой реализации. Нам потребуется лишь одно изменение в момент, когда форма будет отправлена:

```
const submit = event => {
  event.preventDefault();
  onNewColor(titleProps.value, colorProps.value);
  resetTitle();
  resetColor();
};
```

В функции `submit` нам нужно обязательно получить `value` как для заголовка, так и для цвета. Наконец, мы можем применить пользовательские функции сброса, которые были возвращены хуком `useInput`.

Хуки предназначены для использования внутри компонентов React. Мы можем составлять хуки внутри других хуков, потому что в конечном итоге настроенный хук будет использоваться внутри компонента. Изменение состояния в хуке по-прежнему вызывает повторный рендеринг компонента `AddColorForm` с новыми значениями для `titleProps` или `colorProps`.

Добавление цветов в состояние

И контролируемый, и неконтролируемый компонент формы передают значения `title` и `color` родительскому компоненту через функцию `onNewColor`. Родителя не волнует, использован ли управляемый или неконтролируемый компонент — ему нужны только значения для нового цвета.

Добавим любой выбранный нами компонент `AddColorForm` в компонент `App`. Когда вызывается свойство `onNewColor`, мы сохраняем новый цвет в состоянии:

```
import React, { useState } from "react";
import colorData from "./color-data.json";
import ColorList from "./ColorList.js";
import AddColorForm from "./AddColorForm";
import { v4 } from "uuid";

export default function App() {
  const [colors, setColors] = useState(colorData);
  return (
    <>
      <AddColorForm
        onNewColor={(title, color) => {
          const newColors = [
            ...colors,
```

```
            {
              id: v4(),
              rating: 0,
              title,
              color
            }
          ];
          setColors(newColors);
        }}
      />
      <ColorList .../>
    </>
  );
}
```

Когда добавляется новый цвет, вызывается свойство `onNewColor`. Значение `title` и шестнадцатеричное значение нового цвета передаются этой функции в качестве аргументов. Мы используем эти аргументы для создания нового массива цветов. Сначала мы распределяем текущие цвета из состояния в новый массив. Затем добавляем новый объект цвета, используя значения `title` и `color`. Оценка нового цвета равна `0`, потому что мы его пока не оценивали. Мы также используем функцию `v4` из пакета `uuid` для генерации нового уникального `id` цвета. Когда у нас будет массив цветов, содержащий новый цвет, мы сохраним его в состоянии, вызвав функцию `setColors`. Это приведет к повторному рендерингу компонента `App` с новым массивом цветов. Этот новый массив будет использоваться для обновления пользовательского интерфейса. Новый цвет появится внизу списка.

Мы подвели итог первой версии приложения Color Organizer. Теперь пользователи могут добавлять, удалять и оценивать цвета в списке.

Контекст React

Сохранение состояния в одном месте в корне дерева помогло нам добиться большего успеха с ранними версиями React. Любому разработчику на React необходимо научиться передавать состояние как вниз, так и вверх по дереву компонентов через свойства. Однако по мере развития React и роста деревьев компонентов соблюдение этого принципа усложняется. Передача состояния вниз и вверх по дереву через десятки компонентов утомительна и чревата ошибками.

Элементы пользовательского интерфейса, над которыми работает большинство из нас, устроены сложно. Корень дерева часто находится очень далеко от листьев. Получается, что данные, от которых зависит приложение, находятся далеко от компонентов, которые их используют. Каждый компонент получает только те

свойства, которые передает своим потомкам. Это раздувает код и затрудняет масштабирование пользовательского интерфейса.

Передавать данные через огромную череду компонентов в виде свойств до компонента, который должен их использовать, — все равно что ехать поездом из Сан-Франциско в округ Колумбия. Вы проедете через все штаты и не сможете сойти, пока не достигнете пункта назначения (рис. 6.6).

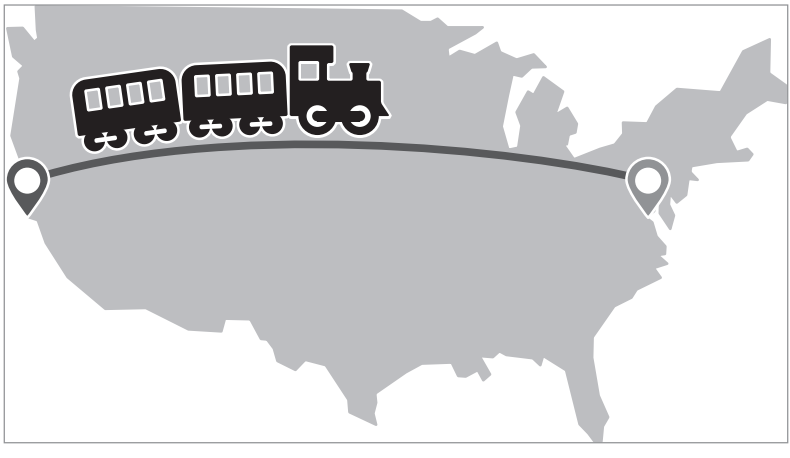


Рис. 6.6. Поезд из Сан-Франциско в округ Колумбия

Очевидно, эффективнее пролететь этот путь на самолете (рис. 6.7).

Рис. 6.7. Перелет из Сан-Франциско в округ Колумбия

В React *контекст* — это отправка данных самолетом. Вы можете разместить данные в контексте React, создав *поставщик контекста* (provider) — компонент, который можно обернуть вокруг всего дерева компонентов или определенных его разделов, другими словами, аэропорт вылета, откуда данные попадают в самолет. Он же — авиационный узел. Все рейсы по разным направлениям отправляются из этого аэропорта. Каждое место назначения является *потребителем контекста* (consumer) — компонентом, который извлекает данные из контекста. Это аэропорт назначения, куда данные приземляются, выгружаются и откуда отправляются в работу.

Использование контекста по-прежнему позволяет хранить данные о состоянии в одном месте, но не требует их передачи через группу компонентов, которым они не нужны.

Размещение цветов в контексте

Чтобы использовать контекст в React, сначала поместим в поставщик контекста некоторые данные и добавим этого поставщика в дерево компонентов. В React есть функция `createContext` для создания нового объекта контекста. Этот объект содержит два компонента: `Provider` и `Consumer`.

Поместим цвета по умолчанию из файла color-data.json в контекст. Добавим контекст в файл index.js — точку входа в приложение:

```
import React, { createContext } from "react";
import colors from "./color-data";
import { render } from "react-dom";
import App from "./App";

export const ColorContext = createContext();

render(
  <ColorContext.Provider value={{ colors }}>
    <App />
  </ColorContext.Provider>,
  document.getElementById("root")
);
```

Используя функцию `createContext`, мы создали новый экземпляр контекста `ColorContext`. Он содержит два компонента: `ColorContext.Provider` и `ColorContext.Consumer`. Поставщик нужен, чтобы поместить цвета в состояние. Мы добавляем данные в контекст, устанавливая свойство `value` компонента `Provider`. В этом сценарии мы добавили в контекст объект, содержащий `colors`. Поскольку мы обернули весь компонент `App` поставщиком, массив `color` будет доступен для любых потребителей контекста в дереве компонентов.

Важно отметить, что мы также экспортировали `ColorContext` из этого места, чтобы получить доступ к `ColorContext.Consumer`, когда мы захотим получить `colors` из контекста.

`ContextProvider` не всегда должен оборачивать все приложение. Вы можете обернуть с его помощью компоненты определенных разделов, если это сделает приложение более эффективным. `Provider` будет предоставлять значения контекста только своим дочерним элементам.

Вы можете использовать несколько поставщиков контекста, иногда неосознанно. Многие пакеты npm, разработанные для работы с React, используют контекст под капотом.

Теперь, когда значение `colors` лежит в контексте, компоненту `App` больше не нужно сохранять состояние и передавать его потомкам в виде свойств. Мы сделали компонент `App` своего рода «эстакадой». `Provider` является родительским компонентом `App` и предоставляет массив `colors` в контексте. `ColorList` является дочерним элементом компонента `App` и может получать `colors` напрямую. Таким образом, приложению вообще не нужно касаться цветов, и это здорово, потому что сам компонент `App` не имеет ничего общего с цветами. Ответственность за цвета делегируется дальше по дереву.

Мы можем удалить из компонента `App` много строк кода. Он должен только отображать `AddColorForm` и `ColorList`. Беспокоиться о данных ему не нужно:

```
import React from "react";
import ColorList from "./ColorList.js";
import AddColorForm from "./AddColorForm";

export default function App() {
  return (
    <>
      <AddColorForm />
      <ColorList />
    </>
  );
}
```

Получение массива colors с помощью useContext

Добавление хуков превращает работу с контекстом в сплошное удовольствие. Хук `useContext` получает нужные нам значения из контекста `Consumer`. Компонент `ColorList` теперь получает массив цветов не из своих свойств, а напрямую через хук `useContext`:

```
import React, { useContext } from "react";
import { ColorContext } from "./";
import Color from "./Color";

export default function ColorList() {
  const { colors } = useContext(ColorContext);
  if (!colors.length) return <div>No Colors Listed. (Add a Color)</div>;
  return (
    <div className="color-list">
      {
        colors.map(color => <Color key={color.id} {...color} />)
      }
    </div>
  );
}
```

Мы изменили компонент `ColorList` и удалили свойство `colors=[]`, потому что массив `colors` извлекается из контекста. Хук `useContext` требует, чтобы экземпляр контекста получил от него значения. Экземпляр `ColorContext` импортируется из файла index.js, в котором мы создаем контекст и добавляем поставщика в дерево компонентов. `ColorList` теперь может создавать пользовательский интерфейс на основе данных, которые были предоставлены в контексте.

ИСПОЛЬЗОВАНИЕ CONSUMER В КОНТЕКСТЕ

Доступ к компоненту `Consumer` осуществляется с помощью хука `useContext`, что означает, что нам больше не нужно работать напрямую с потребителем. Когда не было хуков, приходилось получать цвета из контекста, используя паттерн, называемый *свойствами рендеринга* в потребителе контекста. Свойства рендеринга передаются дочерней функции как аргументы. В следующем примере показано, как использовать потребителя для получения цветов из контекста:

```
export default function ColorList() {
  return (
    <ColorContext.Consumer>
      {context => {
      if (!context.colors.length)
      return <div>No Colors Listed. (Add a Color)</div>;
        return (
          <div className="color-list">
            {
              context.colors.map(color =>
  <Color key={color.id} {...color} />)
            }
          </div>
        )
      }}
    </ColorContext.Consumer>
  )
}
```

Провайдеры контекста с отслеживанием состояния

Поставщик контекста может поместить объект в контекст, но не может самостоятельно изменять значения объектов в контексте. В этом ему нужна помощь со стороны родительского компонента. Необходимо создать компонент с отслеживанием состояния, который отображает поставщика контекста. Когда состояние компонента изменяется, он повторно отображает поставщика контекста с новыми данными контекста. Любые дочерние элементы поставщиков контекста также отображаются повторно с новыми данными контекста.

Компонент с отслеживанием состояния, который отображает поставщика контекста, — это *пользовательский поставщик*. Он будет использоваться, когда нужно обернуть приложение в оболочку поставщика. Создадим `ColorProvider` в новом файле:

```
import React, { createContext, useState } from "react";
import colorData from "./color-data.json";

const ColorContext = createContext();

export default function ColorProvider ({ children }) {
  const [colors, setColors] = useState(colorData);
  return (
    <ColorContext.Provider value={{ colors, setColors }}>
      {children}
    </ColorContext.Provider>
  );
};
```

`ColorProvider` — это компонент, который отображает компонент `ColorContext.Provider`. В нем мы создали переменную состояния для цветов с помощью хука `useState`. Исходные данные массива `colors` все еще берутся из файла color-data.json. Затем `ColorProvide` переносит цвета из состояния в контекст, используя свойство `value` компонента `ColorContext.Provider`. Любые дочерние элементы, отображаемые в `ColorProvider`, будут заключены в `ColorContext.Provider` и получат доступ к массиву `colors` из контекста.

Вы могли заметить, что функция `setColors` тоже добавляется в контекст. Это дает потребителям контекста возможность изменять значение цветов. Всякий раз, когда вызывается функция `setColors`, массив `colors` изменяется. В результате `ColorProvider` отображается заново и пользовательский интерфейс обновляется, выводя новый массив цветов.

Возможно, добавление функции `setColors` в контекст — не лучшая идея. Но это позволяет другим разработчикам и вам в дальнейшем делать ошибки при его

использовании. Когда дело доходит до изменения значения массива `colors`, есть только три варианта: добавление, удаление или оценка цвета. Желательно добавить в контекст функции для каждой из этих операций. То есть вы не открываете функцию `setColors` потребителям, открывая функции только для тех изменений, которые им разрешено делать:

```
export default function ColorProvider ({ children }) {
  const [colors, setColors] = useState(colorData);

  const addColor = (title, color) =>
    setColors([
      ...colors,
      {
        id: v4(),
        rating: 0,
        title,
        color
      }
    ]);

  const rateColor = (id, rating) =>
    setColors(
      colors.map(color => (color.id === id ? { ...color, rating } : color))
    );

  const removeColor = id => setColors(colors.filter(color => color.id !== id));

  return (
    <ColorContext.Provider value={{ colors, addColor, removeColor, rateColor }}>
      {children}
    </ColorContext.Provider>
  );
};
```

Так уже лучше. Мы добавили в контекст функции для всех операций, которые могут быть выполнены с массивом `colors`. Теперь любой компонент в дереве может использовать эти операции и изменять цвета с помощью простых функций, которые можно задокументировать.

Пользовательские хуки с контекстом

Теперь мы можем сделать еще одно изменение. Хуки избавили нас от необходимости открывать контекст для компонентов потребителей. Будем честны: контекст может сбить с толку членов вашей команды, которые не читали эту книгу. Мы можем упростить им задачу, заключив контекст в специальный хук. Вместо открытия экземпляра `ColorContext` создадим хук с именем `useColors`, который будет возвращать цвета из контекста:

```
import React, { createContext, useState, useContext } from "react";
import colorData from "./color-data.json";
import { v4 } from "uuid";

const ColorContext = createContext();
export const useColors = () => useContext(ColorContext);
```

Это простое изменение имеет огромное влияние на архитектуру. Мы объединили все функции, необходимые для рендеринга и работы с цветами с отслеживанием состояния, в один модуль JavaScript. Контекст содержится в этом модуле, но доступен через хук. Это работает, потому что мы возвращаем контекст с помощью хука `useContext`, который имеет доступ к `ColorContext` локально в этом файле. Теперь стоит переименовать этот модуль в color-hooks.js и подарить его функционал сообществу.

Использование цветов с помощью `ColorProvider` и хука `useColors` — сплошное удовольствие. Ради этого стоит заниматься программированием. Применим этот хук к текущему приложению Color Organizer. Сначала обернем компонент приложения пользовательской версией `ColorPro` в файле index.js:

```
import React from "react";
import { ColorProvider } from "./color-hooks.js";
import { render } from "react-dom";
import App from "./App";

render(
  <ColorProvider>
    <App />
  </ColorProvider>,
  document.getElementById("root")
);
```

Теперь любой компонент, дочерний по отношению к `App`, сможет получать данные о цвете из хука `useColors`. Компоненту `ColorList` необходим доступ к массиву `colors` для отображения цветов на экране:

```
import React from "react";
import Color from "./Color";
import { useColors } from "./color-hooks";

export default function ColorList() {
  const { colors } = useColors();
  return ( ... );
}
```

Мы удалили из этого компонента все ссылки на контекст. Все, что ему нужно, теперь доставляется из хука. Компонент `Color` может использовать хук для непосредственного получения функций для оценки и удаления цветов:

```
import React from "react";
import StarRating from "./StarRating";
import { useColors } from "./color-hooks";

export default function Color({ id, title, color, rating }) {
  const { rateColor, removeColor } = useColors();
  return (
    <section>
      <h1>{title}</h1>
      <button onClick={() => removeColor(id)}>X</button>
      <div style={{ height: 50, backgroundColor: color }} />
      <StarRating
        selectedStars={rating}
        onRate={rating => rateColor(id, rating)}
      />
    </section>
  );
}
```

Компоненту `Color` больше не нужно передавать события родительскому объекту через свойства функции. У него есть доступ к функциям `rateColor` и `removeColor` в контексте. Их легко получить с помощью хука `useColors`. Все получилось круто, но мы еще не закончили. Компоненту `AddColorForm` тоже будет полезно использовать хук `useColors`:

```
import React from "react";
import { useInput } from "./hooks";
import { useColors } from "./color-hooks";

export default function AddColorForm() {
  const [titleProps, resetTitle] = useInput("");
  const [colorProps, resetColor] = useInput("#000000");
  const { addColor } = useColors();

  const submit = e => {
    e.preventDefault();
    addColor(titleProps.value, colorProps.value);
    resetTitle();
    resetColor();
  };

  return ( ... );
}
```

Компонент `AddColorForm` может добавлять цвета напрямую с помощью функции `addColor`. Когда цвета добавляются, оцениваются или удаляются, состояние значения массива `colors` в контексте изменяется, а дочерние элементы `ColorProvider` отображаются заново с новыми данными. Все это происходит с помощью простого хука.

Хуки дарят разработчикам мотивацию и удовольствие от программирования через интерфейс. В первую очередь это отличный инструмент для разделения задач. Теперь компонентам React остается думать только об отображении других компонентов React и поддержке пользовательского интерфейса в актуальном состоянии. Хуки React могут заниматься логикой приложения. И пользовательский интерфейс, и хуки можно отдельно разрабатывать, тестировать и даже развертывать. Все это очень уместно в React.

Мы немного коснулись возможностей хуков. В следующей главе мы разберем их подробнее.

ГЛАВА 7

Улучшение компонентов с помощью хуков

Рендеринг (rendering) — это основа приложения React. Когда что-то изменяется (свойства, состояние), дерево компонентов отображается заново, показывая новые данные в виде пользовательского интерфейса. До этого момента нашей рабочей лошадкой для описания способов рендеринга компонентов был хук `useState`. Но есть и другие хуки, определяющие правила рендеринга и повышающие его производительность.

В предыдущей главе мы поговорили о хуках `useState`, `useRef` и `useContext` и увидели, что их можно скомпоновать в новые хуки: `useInput` и `useColors`. Однако мы не все сказали об их происхождении. В React есть много встроенных хуков. В этой главе мы подробно рассмотрим хуки `useEffect`, `useLayoutEffect` и `useReducer`. Все они важны для создания приложений. Мы также рассмотрим хуки `useCallback` и `useMemo`, оптимизирующие компоненты для повышения производительности.

Знакомство с хуком useEffect

Мы получили представление о том, что происходит при рендеринге компонента. Компонент — это просто функция, которая отображает пользовательский интерфейс. Рендеринг происходит при первой загрузке приложения и при изменении значений свойств и состояния. Но что происходит после рендеринга?

Рассмотрим простой компонент `Checkbox`. С помощью хука `useState` установим значение `checked` и функцию `setChecked` для изменения этого значения. Пользователь может устанавливать и снимать флажок, но как предупредить пользователя о том, что флажок установлен? Попробуем сделать это, используя `alert` для блокировки:

```
import React, { useState } from "react";

function Checkbox() {
  const [checked, setChecked] = useState(false);

  alert(`checked: ${checked.toString()}`);

  return (
    <>
      <input
        type="checkbox"
        value={checked}
        onChange={() => setChecked(checked => !checked)}
      />
      {checked ? "checked" : "not checked"}
    </>
  );
};
```

`alert` блокирует рендеринг. Компонент не будет отображаться, пока пользователь не нажмет кнопку OK в окне предупреждения. Поскольку предупреждение является блокирующим, мы не видим следующего состояния отображаемого флажка до нажатия кнопки OK.

Мы тут не за этим, но, может, стоит выдать предупреждение после возврата?

```
function Checkbox {
  const [checked, setChecked] = useState(false);

  return (
    <>
      <input
        type="checkbox"
        value={checked}
        onChange={() => setChecked(checked => !checked)}
      />
      {checked ? "checked" : "not checked"}
    </>
  );

  alert(`checked: ${checked.toString()}`);
};
```

Ладно, забудьте. Мы не можем вызвать `alert` после рендеринга, потому что код вообще не выполнится. Чтобы убедиться, что `alert` отобразится правильно, используем хук `useEffect`. Размещение `alert` внутри функции `useEffect` означает, что функция будет вызываться после рендеринга в качестве побочного эффекта:

```
function Checkbox {
  const [checked, setChecked] = useState(false);
```

```
  useEffect(() => {
    alert(`checked: ${checked.toString()}`);
  });

  return (
    <>
      <input
        type="checkbox"
        value={checked}
        onChange={() => setChecked(checked => !checked)}
      />
      {checked ? "checked" : "not checked"}
    </>
  );
};
```

Мы используем хук `useEffect`, когда рендеринг должен вызвать побочные эффекты. Побочный эффект — это что-то, что функция делает, но не возвращает. Функция `Checkbox` отображает пользовательский интерфейс. Возложим на компонент чуть больше задач. Эти задачи, отличные от возврата пользовательского интерфейса, называются *эффектами*.

`Alert`, `console.log` или взаимодействие с браузером или собственным API не являются частью рендеринга и возврата. Однако в приложении React рендеринг влияет на результаты одного из этих событий. Мы можем использовать хук `useEffect`, чтобы дождаться рендеринга, а затем передать значения в предупреждение или `console.log`:

```
useEffect(() => {
  console.log(checked ? "Yes, checked" : "No, not checked");
});
```

Аналогично мы можем проверить значение `checked` при рендеринге, а затем установить его в `localStorage`:

```
useEffect(() => {
  localStorage.setItem("checkbox-value", checked);
});
```

Мы также можем использовать `useEffect`, чтобы сосредоточиться на конкретном вводе текста, который был добавлен в DOM. React отобразит результат, а затем вызовет `useEffect` для фокусировки элемента:

```
useEffect(() => {
  txtInputRef.current.focus();
});
```

При рендеринге у `txtInputRef` будет значение. Мы можем получить доступ к этому значению в эффекте, чтобы применить фокус. Каждый раз, когда мы

выполняем рендеринг, `useEffect` получает доступ к последним значениям отображения: `props`, `state`, `refs` и т. д.

Хук `useEffect` можно считать функцией, которая выполняется после рендеринга. Когда происходит рендеринг, мы получаем доступ к текущим значениям состояния в компоненте и можем использовать их для чего-то еще. Как только мы снова выполним рендеринг, все начнется заново. Новые значения, новые отображения, новые эффекты.

Массив зависимостей

Хук `useEffect` создан для работы с другими хуками с отслеживанием состояния, такими как `useState` и ранее не упоминавшийся `useReducer`, который мы обсудим позже в этой главе. React повторно отображает дерево компонентов при изменении состояния. Как мы узнали, хук `useEffect` вызывается после рендеринга.

Рассмотрим код, в котором у компонента `App` есть два отдельных значения состояния:

```
import React, { useState, useEffect } from "react";
import "./App.css";

function App() {
  const [val, set] = useState("");
  const [phrase, setPhrase] = useState("example phrase");

  const createPhrase = () => {
    setPhrase(val);
    set("");
  };

  useEffect(() => {
    console.log(`typing "${val}"`);
  });

  useEffect(() => {
    console.log(`saved phrase: "${phrase}"`);
  });

  return (
    <>
      <label>Favorite phrase:</label>
      <input
        value={val}
        placeholder={phrase}
        onChange={e => set(e.target.value)}
      />
```

```
      <button onClick={createPhrase}>send</button>
    </>
  );
}
```

`val` — это переменная состояния, представляющая значение поля ввода. `Val` изменяется всякий раз, когда изменяется значение поля ввода. Это заставляет компонент отображаться каждый раз, когда пользователь вводит новый символ. Когда пользователь нажимает кнопку Отправить, значение `val` текстовой области сохраняется как фраза, а сама `val` сбрасывается до значения `""`, что очищает текстовое поле.

Это работает так, как ожидалось, но хук `useEffect` вызывается больше, чем нужно. После каждого рендеринга вызываются оба хука `useEffect`:

```
typing ""                          // Первый рендеринг
saved phrase: "example phrase"     // Первый рендеринг
typing "S"                         // Второй рендеринг
saved phrase: "example phrase"     // Второй рендеринг
typing "Sh"                        // Третий рендеринг
saved phrase: "example phrase"     // Третий рендеринг
typing "Shr"                       // Четвертый рендеринг
saved phrase: "example phrase"     // Четвертый рендеринг
typing "Shre"                      // Пятый рендеринг
saved phrase: "example phrase"     // Пятый рендеринг
typing "Shred"                     // Шестой рендеринг
saved phrase: "example phrase"     // Шестой рендеринг
```

Мы не хотим, чтобы каждый эффект запускался при каждом рендеринге. Нам нужно связать хуки `useEffect` с конкретными изменениями данных. Применим массив зависимостей для управления моментом вызова эффекта:

```
useEffect(() => {
  console.log(`typing "${val}"`);
}, [val]);

useEffect(() => {
  console.log(`saved phrase: "${phrase}"`);
}, [phrase]);
```

Мы добавили массив зависимостей к обоим эффектам, чтобы управлять их запуском. Первый эффект вызывается только при изменении значения `val`. Второй — только при изменении значения `phrase`. Когда мы запустим приложение и посмотрим в консоль, то увидим более эффективные обновления:

```
typing ""                          // Первый рендеринг
saved phrase: "example phrase"     // Первый рендеринг
typing "S"                         // Второй рендеринг
typing "Sh"                        // Третий рендеринг
```

```
typing "Shr"                        // Четвертый рендеринг
typing "Shre"                       // Пятый рендеринг
typing "Shred"                      // Шестой рендеринг
typing ""                           // Седьмой рендеринг
saved phrase: "Shred"               // Седьмой рендеринг
```

Изменение значения `val` путем ввода текста вызывает срабатывание только первого эффекта. Когда мы нажимаем кнопку, значение `phrase` сохраняется, а значение `val` сбрасывается до `""`.

Это массив, поэтому мы можем проверить несколько его значений. Допустим, мы хотим запускать определенный эффект каждый раз, когда меняются значения `val` или `phrase`:

```
useEffect(() => {
  console.log("either val or phrase has changed");
}, [val, phrase]);
```

Если любое из этих значений изменится, эффект будет вызван снова. Также можно в качестве второго аргумента функции `useEffect` указать пустой массив зависимостей. Это приведет к тому, что эффект будет вызываться только один раз после первого рендеринга:

```
useEffect(() => {
  console.log("only once after initial render");
}, []);
```

Поскольку в массиве нет зависимостей, эффект вызывается в момент начального рендеринга. Отсутствие зависимостей означает отсутствие изменений, поэтому больше эффект вызываться не будет. Эффекты, которые вызываются только при первом рендеринге, чрезвычайно полезны для инициализации:

```
useEffect(() => {
  welcomeChime.play();
}, []);
```

Если вы вернете функцию из эффекта, она будет вызываться при удалении компонента из дерева:

```
useEffect(() => {
  welcomeChime.play();
  return () => goodbyeChime.play();
}, []);
```

Это означает, что вы можете использовать хук `useEffect` для настройки и разборки. Пустой массив означает, что приветственный звуковой сигнал будет воспроизводиться один раз при первом рендеринге. Затем мы вернем функцию

как функцию очистки, воспроизводя прощальный звуковой сигнал, когда компонент будет удален из дерева.

Этот паттерн часто бывает полезен. Представьте, что мы подписываемся на ленту новостей при первом рендеринге, а затем отписываемся от ленты с помощью функции очистки. Начнем с создания значения состояния для сообщений и функции `setPosts` для изменения этого значения. Затем создадим функцию `addPosts`, которая будет принимать самый последний пост и добавлять его в массив. После этого используем хук `useEffect`, чтобы подписаться на новостную ленту и воспроизвести звуковой сигнал. Дополнительно вернем функции очистки, отписки и воспроизведения прощального звукового сигнала:

```
const [posts, setPosts] = useState([]);
const addPost = post => setPosts(allPosts => [post, ...allPosts]);

useEffect(() => {
  newsFeed.subscribe(addPost);
  welcomeChime.play();
  return () => {
    newsFeed.unsubscribe(addPost);
    goodbyeChime.play();
  };
}, []);
```

Хук `useEffect` перегружен работой. Попробуем использовать два отдельных `useEffect`: для событий новостной ленты и для событий, связанных со звуковым сигналом:

```
useEffect(() => {
  newsFeed.subscribe(addPost);
  return () => newsFeed.unsubscribe(addPost);
}, []);

useEffect(() => {
  welcomeChime.play();
  return () => goodbyeChime.play();
}, []);
```

Разделение функциональности на несколько вызовов `useEffect` обычно полезно. Но давайте сделаем еще лучше. Создадим функциональность для подписки на новостную ленту. В момент подписки, отказа от подписки и всякий раз при появлении нового поста будут воспроизводиться джазовые звуки. Все ведь любят громкие звуки, верно? Нам понадобится пользовательский хук. Назовем его `useJazzyNews`:

```
const useJazzyNews = () => {
  const [posts, setPosts] = useState([]);
```

```
  const addPost = post => setPosts(allPosts => [post, ...allPosts]);

  useEffect(() => {
    newsFeed.subscribe(addPost);
    return () => newsFeed.unsubscribe(addPost);
  }, []);

  useEffect(() => {
    welcomeChime.play();
    return () => goodbyeChime.play();
  }, []);

  return posts;
};
```

Этот пользовательский хук содержит все функции для обработки ярких новостей, а это означает, что мы можем легко поделиться этой функцией с компонентами, например с новым компонентом `NewsFeed`:

```
function NewsFeed({ url }) {
  const posts = useJazzyNews();

  return (
    <>
      <h1>{posts.length} articles</h1>
      {posts.map(post => (
        <Post key={post.id} {...post} />
      ))}
    </>
  );
}
```

Глубокая проверка зависимостей

Сейчас зависимости, которые мы добавили в массив, представлены строками. Такие примитивы JavaScript, как строки, логические значения, числа и прочие, можно сравнивать. Строка должна соответствовать ожидаемой строке:

```
if ("gnar" === "gnar") {
  console.log("gnarly!!");
}
```

Однако сравнивать объекты, массивы и функции не так просто. Например, сравним два массива:

```
if ([1, 2, 3] !== [1, 2, 3]) {
  console.log("but they are the same");
}
```

Массивы `[1,2,3]` и `[1,2,3]` не равны, даже если они одинаковы по длине и содержимому. Дело в том, что это два разных экземпляра массива. Если мы создадим переменную для хранения этого значения массива, а затем выполним сравнения, то увидим ожидаемый результат:

```
const array = [1, 2, 3];
if (array === array) {
  console.log("because it's the exact same instance");
}
```

В JavaScript массивы, объекты и функции равны только в случае, если они являются одним и тем же экземпляром. Причем здесь массив зависимостей `useEffect`? Возьмем компонент, который мы можем заставить выполнять рендеринг столько, сколько захотим. Создадим хук, который заставляет компонент рендериться при каждом нажатии клавиши:

```
const useAnyKeyToRender = () => {
  const [, forceRender] = useState();

  useEffect(() => {
    window.addEventListener("keydown", forceRender);
    return () => window.removeEventListener("keydown", forceRender);
  }, []);
};
```

Все, что нам нужно сделать для принудительного рендеринга, — вызвать функцию изменения состояния. Значение состояния нас не волнует. Нам нужна только функция состояния `forceRender` (вот почему мы добавили запятую, используя деструктуризацию массива, — вспомните главу 2). Когда компонент отображается впервые, мы услышим событие нажатия на клавишу. Когда будет нажата клавиша, мы заставим компонент отобразиться, вызвав `forceRender`. Как и раньше, мы вернем функцию очистки, в которой перестанем прослушивать события нажатия клавиш. Добавив этот хук к компоненту, мы можем принудительно выполнить повторный рендеринг, просто нажав клавишу.

Создав пользовательский хук, мы можем применить его в компоненте `App` (и в любом другом компоненте; хуки — это круто):

```
function App() {
  useAnyKeyToRender();

  useEffect(() => {
    console.log("fresh render");
  });

  return <h1>Open the console</h1>;
}
```

Каждый раз, когда мы нажимаем клавишу, отображается компонент `App`. Хук `useEffect` выводит сообщение «fresh render» в консоль каждый раз, когда приложение отображается. Настроим хук `useEffect` в компоненте `App` так, чтобы он ссылался на значение `word`. Если `word` изменится, выполнится повторный рендеринг:

```
const word = "gnar";
useEffect(() => {
  console.log("fresh render");
}, [word]);
```

Мы будем вызывать `useEffect` не для каждого события нажатия клавиши, а только после первого рендеринга и при каждом изменении значения `word`. Поскольку оно не меняется, повторных отображений не происходит. Добавление примитива или числа в массив зависимостей работает правильно. Эффект активируется один раз.

Что произойдет, если вместо одного слова мы используем массив слов?

```
const words = ["sick", "powder", "day"];
useEffect(() => {
  console.log("fresh render");
}, [words]);
```

`words` — это массив. Поскольку новый массив объявляется при каждом рендеринге, JavaScript предполагает, что слова изменяются при каждом вызове эффекта «fresh render». Каждый раз массив получается новым, и это регистрируется как обновление, которое должно запускать рендеринг.

Объявление массива `words` за пределами компонента `App` решит проблему:

```
const words = ["sick", "powder", "day"];

function App() {
  useAnyKeyToRender();
  useEffect(() => {
    console.log("fresh render");
  }, [words]);

  return <h1>component</h1>;
}
```

Массив зависимостей в этом случае — это экземпляр `words`, объявленный вне функции. Эффект «fresh render» больше не вызывается после первого рендеринга, потому что `words` является все тем же экземпляром. Это хорошее решение для данного примера, но не всегда можно (и нежелательно) определять переменную вне области видимости функции. Иногда значение, передаваемое в массив

зависимостей, требует наличия переменных в области видимости. Например, создадим массив `words` из свойства `children`:

```
function WordCount({ children = "" }) {
  useAnyKeyToRender();

  const words = children.split(" ");

  useEffect(() => {
    console.log("fresh render");
  }, [words]);

  return (
    <>
      <p>{children}</p>
      <p>
        <strong>{words.length} — words</strong>
      </p>
    </>
  );
}

function App() {
  return <WordCount>You are not going to believe this but...</WordCount>;
}
```

Компонент `App` содержит слова, которые являются дочерними по отношению к компоненту `WordCount`, который принимает `children` как свойство. Мы устанавливаем `words` в компоненте равными массиву тех слов, на которых мы выполняли метод `.split`. Мы надеемся, что компонент будет выполнять повторный рендеринг только в случае изменения `words`, но как только мы нажимаем клавишу, то снова видим страшную фразу «fresh render».

Но бояться нечего, потому что команда React предоставила нам способ избежать лишних рендерингов. Не оставлять же нас в беде. Решением этой проблемы, как и следовало ожидать, стал хук — `useMemo`.

Он вызывает функцию вычисления мемоизированного значения. В computer science мемоизация — это технология повышения производительности. В мемоизированной функции результат вызова функции сохраняется и кешируется. Затем, когда функция вызывается снова с теми же входными данными, вместо повторно вычисленного значения возвращается кешированное. Хук `useMemo` позволяет сравнивать кешированное значение с новым результатом.

Хук `UseMemo` работает так: мы передаем ему функцию, которая используется для вычисления и создания мемоизированного значения. `useMemo` будет пересчитывать это значение только при изменении одной из зависимостей функции. Сначала импортируем хук `useMemo`:

```
import React, { useEffect, useMemo } from "react";
```

Затем воспользуемся функцией, чтобы задать `words`:

```
const words = useMemo(() => {
  const words = children.split(" ");
  return words;
}, []);

useEffect(() => {
  console.log("fresh render");
}, [words]);
```

Хук `useMemo` вызывает переданную ему функцию и устанавливает `words` в значение, возвращаемое этой функцией. Как и `useEffect`, `useMemo` использует массив зависимостей:

```
const words = useMemo(() => children.split(" "));
```

Если мы не включаем массив зависимостей с помощью `useMemo`, слова вычисляются при каждом рендеринге. Массив зависимостей определяет, когда функция обратного вызова должна вызываться. Второй аргумент, передаваемый функции `useMemo`, — это массив зависимостей, который должен содержать значение `children`:

```
function WordCount({ children = "" }) {
  useAnyKeyToRender();

  const words = useMemo(() => children.split(" "), [children]);

  useEffect(() => {
    console.log("fresh render");
  }, [words]);

  return (...);
}
```

Массив `words` зависит от свойства `children`. Если значение `children` изменяется, нужно вычислить новое значение `words`, которое отражает это изменение. В этот момент `useMemo` вычисляет новое значение `words` при первом рендеринге компонента и при изменении свойства `children`.

При создании приложений React очень важно разобраться с работой хука `useMemo`.

Хук `useCallback` можно использовать как `useMemo`, но он запоминает функции, а не значения. Например:

```
const fn = () => {
  console.log("hello");
  console.log("world");
```

```
};

useEffect(() => {
  console.log("fresh render");
  fn();
}, [fn]);
```

`fn` — это функция, которая записывает «Hello», а затем «World». Это зависимость `useEffect`, но, как и в случае с `words`, JavaScript предполагает, что в каждом отображении `fn` отличается и запускает эффект при новом рендеринге. Наш «fresh render» появляется при нажатии клавиши. Так не пойдет.

Начнем с обертывания функции с помощью хука `useCallback`:

```
const fn = useCallback(() => {
  console.log("hello");
  console.log("world");
}, []);

useEffect(() => {
  console.log("fresh render");
  fn();
}, [fn]);
```

Хук `useCallback` запоминает значение функции `fn`. Так же, как `useMemo` и `useEffect`, он ожидает в качестве второго аргумента массив зависимостей. Мы создадим мемоизированный обратный вызов один раз, потому что массив зависимостей пуст.

Теперь, когда у нас есть понимание использования и различий между `useMemo` и `useCallback`, улучшим хук `useJazzyNews`. Каждый раз, когда появляется новое сообщение, мы будем вызывать метод `newPostChime.play ()`. В этом хуке `posts` — это массив, необходимый для того, чтобы `useMemo` запомнил значения:

```
const useJazzyNews = () => {
  const [_posts, setPosts] = useState([]);
  const addPost = post => setPosts(allPosts => [post, ...allPosts]);

  const posts = useMemo(() => _posts, [_posts]);

  useEffect(() => {
    newPostChime.play();
  }, [posts]);

  useEffect(() => {
    newsFeed.subscribe(addPost);
    return () => newsFeed.unsubscribe(addPost);
  }, []);

  useEffect(() => {
```

```
    welcomeChime.play();
    return () => goodbyeChime.play();
  }, []);

  return posts;
};
```

Теперь хук `useJazzyNews` воспроизводит звуковой сигнал каждый раз, когда появляется новое сообщение, благодаря замене `const[posts, setPosts]` на `const[_posts, setPosts]`. Мы будем вычислять новое значение для сообщений при каждом изменении `_posts`.

Еще мы добавили эффект, который воспроизводит звуковой сигнал при каждом изменении массива `post`. Мы прослушиваем новостную ленту на предмет новых сообщений. Когда добавляется новый пост, хук повторно вызывается с `_posts`, отображая новое сообщение. Затем новое значение `post` запоминается, потому что `_posts` изменилось, и звучит сигнал, потому что этот эффект зависит от `posts`. Он воспроизводится только при изменении сообщений, а список сообщений изменяется только при добавлении нового.

Позже в этой главе мы обсудим React Profiler — расширение браузера для тестирования производительности и рендеринга компонентов React. И подробно рассмотрим, когда использовать `useMemo` и `useCallback`. (Спойлер: без фанатизма!)

Когда использовать LayoutEffect

Мы понимаем, что рендеринг всегда предшествует `useEffect`. Сначала выполняется рендеринг, затем по порядку запускаются эффекты с полным доступом ко всем значениям рендеринга. Беглый взгляд на документацию React покажет, что есть еще один хук эффектов — `useLayoutEffect`.

Хук `useLayoutEffect` вызывается в определенный момент цикла рендеринга. Последовательность событий следующая:

1. Рендеринг.
2. Вызов `useLayoutEffect`.
3. Отрисовка браузера: момент, когда элементы компонента фактически добавляются в DOM.
4. Вызов `useEffect`.

Это можно увидеть, добавив в консоль несколько простых сообщений:

```
import React, { useEffect, useLayoutEffect } from "react";

function App() {
```

```
  useEffect(() => console.log("useEffect"));
  useLayoutEffect(() => console.log("useLayoutEffect"));
  return <div>ready</div>;
}
```

В компоненте App первым перехватчиком является useEffect, за которым следует useLayoutEffect. Мы видим, что useLayoutEffectis вызывается перед useEffect:

```
useLayoutEffect
useEffect
```

useLayoutEffect вызывается после рендеринга, но до того, как браузер отображает изменение. В большинстве случаев useEffect подходит для работы, но если эффект важен для отрисовки браузера (его внешнего вида или размещения элементов пользовательского интерфейса на экране), вы можете использовать useLayoutEffect. Например, узнаем ширину и высоту элемента при изменении размера окна:

```
function useWindowSize {
  const [width, setWidth] = useState(0);
  const [height, setHeight] = useState(0);

  const resize = () => {
    setWidth(window.innerWidth);
    setHeight(window.innerHeight);
  };

  useLayoutEffect(() => {
    window.addEventListener("resize", resize);
    resize();
    return () => window.removeEventListener("resize", resize);
  }, []);

  return [width, height];
};
```

Ширина и высота окна — это информация, которая может понадобиться компоненту перед тем как браузер начнет отрисовку. Хук useLayoutEffect используется для вычисления ширины и высоты окна перед отображением. Другой пример использования useLayoutEffect — отслеживание положения мыши:

```
function useMousePosition {
  const [x, setX] = useState(0);
  const [y, setY] = useState(0);

  const setPosition = ({ x, y }) => {
    setX(x);
    setY(y);
```

```
  };

  useLayoutEffect(() => {
    window.addEventListener("mousemove", setPosition);
    return () => window.removeEventListener("mousemove", setPosition);
  }, []);

  return [x, y];
};
```

Весьма вероятно, что при отрисовке экрана будут использоваться координаты `x` и `y` мыши. `useLayoutEffect` позволяет предварительно точно их рассчитать.

Правила работы с хуками

При работе с хуками следует помнить о нескольких рекомендациях, которые помогут избежать ошибок и неожиданного поведения:

Хуки работают только в рамках компонента

Хуки следует вызывать только из компонентов React. Их также можно добавлять в пользовательские хуки, которые в конечном итоге добавляются в компоненты. Хуки — это не обычный JavaScript, а паттерн React, но их уже начали моделировать и включать в другие библиотеки.

Лучше разбить функциональность на несколько хуков

В предыдущем примере с компонентом `JazzyNews` мы выделили все, что связано с подписками, в один эффект, а все, что связано со звуком, — в другой. Код стало проще читать, но это еще не все. Поскольку хуки вызываются по порядку, рекомендуется делать их небольшими. После вызова React сохраняет значения хуков в массив, чтобы затем их отслеживать. Рассмотрим компонент:

```
function Counter() {
  const [count, setCount] = useState(0);
  const [checked, toggle] = useState(false);

  useEffect(() => {
    ...
  }, [checked]);

  useEffect(() => {
    ...
  }, []);

  useEffect(() => {
    ...
```

```
  }, [count]);

  return ( ... )
}
```

Порядок вызовов хуков одинаковый для каждого отображения:

```
[count, checked, DependencyArray, DependencyArray, DependencyArray]
```

Хуки следует вызывать только на верхнем уровне

Используйте хуки только на верхнем уровне функции React. Их нельзя помещать в условные операторы, циклы или вложенные функции. Настроим счетчик:

```
function Counter() {
  const [count, setCount] = useState(0);

  if (count > 5) {
    const [checked, toggle] = useState(false);
  }

  useEffect(() => {
    ...
  });

  if (count > 5) {
    useEffect(() => {
      ...
    });
  }

  useEffect(() => {
    ...
  });

  return ( ... )
}
```

Когда мы используем хук `useState` в операторе `if`, то говорим, что хук должен вызываться, если значение счетчика больше `5`. Это сбросит значения массива. Иногда массив будет таким: `[count, checked, DependencyArray, 0, DependencyArray]`. А иногда таким: `[count, DependencyArray, 1]`. Индекс эффекта в массиве важен для React. Значения хранятся именно так.

Но значит ли это, что нельзя использовать условную логику в приложениях React? Конечно, нет! Но нужно правильно организовать условные выражения. Можно вложить операторы `if`, циклы и другие условные выражения в хук:

```
function Counter() {
  const [count, setCount] = useState(0);
  const [checked, toggle] =
  useState(
  count => (count < 5)
  ? undefined
  : !c,
  (count < 5) ? undefined
  );

  useEffect(() => {
    ...
  });

  useEffect(() => {
    if (count < 5) return;
    ...
  });

  useEffect(() => {
    ...
  });

  return ( ... )
}
```

Здесь значение `checked` основано на условии, что `count` больше `5`. Если `count` меньше `5`, значение для `checked` не определено. Вложение этого условия внутрь хука означает, что хук остается на верхнем уровне, но результат при этом аналогичен. Второй эффект работает по тем же правилам. Если `count` меньше `5`, оператор `return` предотвратит продолжение выполнения эффекта. Тогда массив значений хука останется нетронутым: `[countValue, checkedValue, DependencyArray, DependencyArray, DependencyArray]`.

Как и условную логику, асинхронное поведение нужно описывать внутри хука. Хук `useEffect` принимает в качестве первого аргумента функцию, а не промис. Таким образом, не получится использовать асинхронную функцию в качестве первого аргумента: `useEffect(async () => {})`. Однако можно создать асинхронную функцию внутри вложенной функции:

```
useEffect(() => {
  const fn = async () => {
    await SomePromise();
  };
  fn();
});
```

Мы создали переменную `fn` для обработки `async` и `await`, а затем вызвали функцию в качестве возврата. При желании можно дать этой функции имя:

```
  useEffect(() => {
    (async () => {
      await SomePromise();
    })();
  });
```

Следуя этим правилам, вы сможете избежать некоторых распространенных ошибок, связанных с хуками в React. В комплект Create React входит подключаемый модуль ESLint под названием eslint-plugin-react-hooks, который предоставляет предупреждающие подсказки, если вы нарушаете эти правила.

Улучшение кода с помощью хука useReducer

Рассмотрим компонент `Checkbox`, хранящий простое состояние. Галочка либо отмечена, либо нет. `checked` — это значение состояния, а `setChecked` — это функция, которая будет использоваться для изменения состояния. При первом рендеринге компонента значение `checked` будет ложным:

```
function Checkbox() {
  const [checked, setChecked] = useState(false);

  return (
    <>
      <input
        type="checkbox"
        value={checked}
        onChange={() => setChecked(checked => !checked)}
      />
      {checked ? "checked" : "not checked"}
    </>
  );
}
```

Код работает хорошо, но одна часть этой функции может вызывать тревогу:

```
onChange={() => setChecked(checked => !checked)}
```

Посмотрите внимательно. На первый взгляд все нормально, но нет ли здесь подвоха? Мы отправляем функцию, которая принимает текущее значение `checked` и возвращает противоположное значение `!checked`. Выглядит сложнее, чем хотелось бы. Разработчики могут легко отправить неверную информацию и все сломать. Почему бы не сделать функцию в виде переключателя?

Добавим функцию `toggle`, которая будет делать то же самое: вызывать `setChecked` и возвращать значение, противоположное текущему значению `checked`:

```
function Checkbox() {
  const [checked, setChecked] = useState(false);
```

```
  function toggle() {
    setChecked(checked => !checked);
  }

  return (
    <>
      <input type="checkbox" value={checked} onChange={toggle} />
      {checked ? "checked" : "not checked"}
    </>
  );
}
```

Стало лучше. Функция `onChange` имеет предсказуемое значение в виде функции `toggle`. Можно сделать еще один шаг, чтобы получать более предсказуемые результаты каждый раз, когда мы используем компонент `check box`. Помните функцию, которую мы отправили в `setChecked` в функции переключения?

```
setChecked(checked => !checked);
```

Теперь мы будем называть функции, подобные `checked => !checked`, *редукторами*. Самое простое определение функции-редуктора заключается в том, что она принимает текущее состояние и возвращает новое. Если `check` равна `false`, то вернется `true`. Вместо того чтобы жестко кодировать это поведение в событиях `onChange`, мы можем абстрагировать логику до функции-редуктора, которая всегда будет давать одни и те же результаты. Вместо хука `useState` в компоненте мы используем `useReducer`:

```
function Checkbox() {
  const [checked, toggle] = useReducer(checked => !checked, false);

  return (
    <>
      <input type="checkbox" value={checked} onChange={toggle} />
      {checked ? "checked" : "not checked"}
    </>
  );
}
```

Хук `useReducer` принимает функцию-редуктор и исходное состояние `false`. Установим функцию `onChange` на `setChecked`, которая вызовет функцию-редуктор.

Предыдущий редуктор `checked =>!checked` является ярким примером работы редуктора. Если функции предоставляются одни и те же входные данные, следует ожидать одинаковые выходные данные. Эта концепция происходит от метода `Array.reduce` в JavaScript. По сути `reduce` делает то же самое, что и `reducer`: принимает функцию (сводя все значения до одного значения) и начальное значение и возвращает одно значение.

Метод `Array.reduce` принимает функцию-редуктор и начальное значение. Редуктор вызывается для каждого значения `numbers` до тех пор, пока не будет возвращено одно значение:

```
const numbers = [28, 34, 67, 68];

numbers.reduce((number, nextNumber) => number + nextNumber, 0); // 197
```

Редуктор, отправленный в `Array.reduce`, принимает два аргумента. Также функции-редуктору можно отправить несколько аргументов:

```
function Numbers() {
  const [number, setNumber] = useReducer(
    (number, newNumber) => number + newNumber,
    0
  );
  return <h1 onClick={() => setNumber(30)}>{number}</h1>;
}
```

Каждый раз нажимая на `h1`, мы добавляем `30` к общей сумме.

Использование хука useReducer для обработки сложного состояния

Хук `useReducer` помогает предсказуемо обрабатывать и более сложные состояния. Рассмотрим объект, содержащий данные пользователя:

```
const firstUser = {
  id: "0391-3233-3201",
  firstName: "Bill",
  lastName: "Wilson",
  city: "Missoula",
  state: "Montana",
  email: "bwilson@mtnwilsons.com",
  admin: false
};
```

Также у нас есть компонент с именем `User`, который устанавливает `firstUser` в качестве начального состояния и отображает соответствующие данные:

```
function User() {
  const [user, setUser] = useState(firstUser);

  return (
    <div>
      <h1>
        {user.firstName} {user.lastName} – {user.admin ? "Admin" : "User"}
      </h1>
```

```
      <p>Email: {user.email}</p>
      <p>
        Location: {user.city}, {user.state}
      </p>
      <button>Make Admin</button>
    </div>
  );
}
```

Распространенная ошибка при управлении состояниями — перезапись состояния:

```
<button
  onClick={() => {
    setUser({ admin: true });
  }}
>
  Make Admin
</button>
```

Это приведет к перезаписи состояния из `firstUser` и его замене тем, что мы отправили в функцию `setUser` — `{admin: true}`. Это можно исправить, передав текущие значения от пользователя, а затем перезаписав значение `admin`:

```
<button
  onClick={() => {
    setUser({ ...user, admin: true });
  }}
>
  Make Admin
</button>
```

Этот код примет исходное состояние и вставит новые пары ключ — значение: `{admin: true}`. Необходимость переписать эту логику в каждом событии `onClick` приведет к ошибкам (можно просто забыть сделать это):

```
function User() {
  const [user, setUser] = useReducer(
    (user, newDetails) => ({ ...user, ...newDetails }),
    firstUser
  );
  ...
}
```

Затем мы отправим новое значение состояния `newDetails` редуктору, и оно будет помещено в объект:

```
<button
  onClick={() => {
    setUser({ admin: true });
```

```
  }}
>
  Make Admin
</button>
```

Этот паттерн полезен, когда состояние имеет несколько подзначений или следующее состояние зависит от предыдущего. Дай человеку оператор распространения, и он будет доволен один день. Научи человека пользоваться `Reducer`, и он будет счастлив всю жизнь.

УСТАРЕВШИЕ ФУНКЦИИ SETSTATE И USEREDUCER

В предыдущих версиях React для обновления состояния мы использовали функцию `setState`. Первоначальное состояние назначалось в конструкторе как объект:

```
class User extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      id: "0391-3233-3201",
      firstName: "Bill",
      lastName: "Wilson",
      city: "Missoula",
      state: "Montana",
      email: "bwilson@mtnwilsons.com",
      admin: false
    };
  }
}
<button onSubmit={() =>
        {this.setState({admin: true });}}
Make Admin
</button>
```

Раньше `setState` объединяла значения состояния. И `useReducer` тоже:

```
const [state, setState] = useReducer(
        (state, newState) =>
              ({...state, ...newState}),
        initialState);

<button onSubmit={() =>
        {setState({admin: true });}}
Make Admin
</button>
</div>);
```

Если вам нравится этот паттерн, вы можете использовать `legacy-set-state` npm или `useReducer`.

Последние несколько примеров представляют собой простые приложения с редукторами. В следующей главе мы подробнее рассмотрим паттерны проектирования редукторов, которые можно использовать для упрощения управления состоянием в приложениях.

Повышение производительности рендеринга компонентов

В приложении React отображаются компоненты, и обычно их много. Повышение производительности — это устранение лишних элементов и сокращение времени, необходимого на рендеринг. В React есть инструменты, которые могут предотвратить ненужную визуализацию: `memo`, `useMemo` и `useCallback`. Мы рассмотрели `useMemo` и `useCallback` ранее в этой главе, но в этом разделе мы более подробно рассмотрим, как использовать эти хуки.

Функция `memo` используется для создания чистых компонентов. Из главы 3 мы знаем, что при одинаковых параметрах чистая функция всегда будет возвращать один и тот же результат. Чистый компонент работает точно так же. В React чистым компонентом называется компонент, который при одних и тех же свойствах всегда отображает один и тот же вывод.

Создадим компонент под названием `Cat`:

```
const Cat = ({ name }) => {
  console.log(`rendering ${name}`);
  return <p>{name}</p>;
};
```

`Cat` — чистый компонент. Нам всегда выводится абзац, в котором отображается свойство `name`. Если имя, указанное в качестве свойства, совпадает, вывод будет таким же:

```
function App() {
  const [cats, setCats] = useState(["Biscuit", "Jungle", "Outlaw"]);
  return (
    <>
      {cats.map((name, i) => (
        <Cat key={i} name={name} />
      ))}
      <button onClick={() => setCats([...cats, prompt("Name a cat")])}>
        Add a Cat
      </button>
    </>
  );
}
```

Это приложение использует компонент `Cat`. После первого рендеринга в консоли появляются подсказки:

```
rendering Biscuit
rendering Jungle
rendering Outlaw
```

При нажатии кнопки Add a Cat пользователю предлагается добавить кота.

Если мы добавим кота с кличкой «Ripple», мы увидим, что все компоненты `Cat` отображаются заново:

```
rendering Biscuit
rendering Jungle
rendering Outlaw
rendering Ripple
```

Этот код работает за счет подсказок, но это всего лишь пример. Не используйте подсказки в реальном приложении.

Каждый раз, когда мы добавляем нового кота, компонент `Cat` рендерится заново, но мы не хотим повторно отображать чистый компонент, если его свойства не изменились. Функция `memo` позволяет создать компонент, который будет отображаться только при изменении его свойств. Импортируем его из библиотеки React и обернем им текущий компонент `Cat`:

```
import React, { useState, memo } from "react";
const Cat = ({ name }) => {
  console.log(`rendering ${name}`);
  return <p>{name}</p>;
};

const PureCat = memo(Cat);
```

Мы создали новый компонент под названием `PureCat`, который запускает рендеринг `Cat` только при изменении свойств. Затем мы можем заменить компонент `Cat` на `PureCat` в компоненте `App`:

```
cats.map((name, i) => <PureCat key={i} name={name} />);
```

Теперь каждый раз, когда мы добавляем новую кличку, в консоли происходит один рендеринг:

```
rendering Pancake
```

Поскольку клички других котов не изменились, их рендеринг не выполняется. Это хорошо работает для свойства `name`, но что, если мы добавим в компонент `Cat` свойство-функцию?

```
const Cat = memo(({ name, meow = f => f }) => {
  console.log(`rendering ${name}`);
  return <p onClick={() => meow(name)}>{name}</p>;
});
```

Каждый раз, когда пользователь щелкает по кличке кота, можно использовать это свойство, чтобы выводить в консоль мяуканье:

```
<PureCat key={i} name={name} meow={name => console.log(`${name} has meowed`)} />
```

После этого нововведения компонент `PureCat` стал работать неправильно. Теперь он всегда рендерит компонент `Cat`, даже если свойство `name` не меняется. К сожалению, каждый раз, когда мы определяем свойство `meow` как функцию, оно становится новой функцией. Для React свойство `meow` изменилось, и компонент отобразился повторно.

Функция `memo` позволит точнее задать правила рендеринга компонента:

```
const RenderCatOnce = memo(Cat, () => true);
const AlwaysRenderCat = memo(Cat, () => false);
```

Второй аргумент, передаваемый в функцию `memo`, — предикат. Предикат — это функция, которая возвращает истину или ложь. Эта функция решает, нужно ли повторно отображать кличку кота. Только когда предикат возвращает `false`, `Cat` отображается заново. В любом случае `Cat` всегда отображается хотя бы один раз. Вот почему в случае с `RenderCatOnce` он будет отображаться один раз, а затем никогда. Обычно эта функция используется для проверки фактических значений:

```
const PureCat = memo(
  Cat,
  (prevProps, nextProps) => prevProps.name === nextProps.name
);
```

Мы можем использовать второй аргумент, чтобы сравнить свойства и решить, следует ли заново рендерить `Cat`. Предикат получает предыдущие и следующие свойства. Эти объекты используются для проверки свойства `name`. Если оно изменилось, произойдет повторный рендеринг компонента. Если не изменилось, рендеринг будет повторяться независимо от того, что React думает о свойстве `meow`.

shouldComponentUpdate и PureComponent

Введенные нами концепции не новы для React. Функция `memo` — это новое решение давно поставленной проблемы. В предыдущих версиях React был метод `shouldComponentUpdate`. Будучи введенным в компоненте, он использовался, чтобы сообщить React, при каких обстоятельствах компонент должен обновляться. Он описывал, какие свойства или состояние необходимо изменить для повторного рендеринга компонента. Когда-то функция `shouldComponentUpdate` вошла в состав библиотеки React, и многие восприняли ее как настолько по-

лезную, что команда React решила создать альтернативный способ создания компонента как класса. Компонент-класс выглядит так:

```
class Cat extends React.Component {
  render() {
    return (
      {name} is a good cat!
    )
  }
}
```

`PureComponent` выглядит так:

```
class Cat extends React.PureComponent {
  render() {
    return (
      {name} is a good cat!
    )
  }
}
```

`PureComponent` — то же самое, что `React.memo`, но `PureComponens` используется только для компонентов-классов, а `React.memo` — только для компонентов-функций.

Хуки `useCallback` и `useMemo` можно использовать для запоминания свойств объектов и функций. Воспользуемся `useCallback` в компоненте `Cat`:

```
const PureCat = memo(Cat);
function App() {
  const meow = useCallback(name => console.log(`${name} has meowed`, []);
  return <PureCat name="Biscuit" meow={meow} />
}
```

Мы не предоставили в `memo(Cat)` предикат проверки свойств, а использовали `useCallback`, проверяющий, что функция `meow` не изменилась. Использование этих функций может быть полезно при работе со слишком большим количеством повторных отображений в дереве компонентов.

Когда проводить рефакторинг

Последние рассмотренные нами хуки `useMemo` и `useCallback`, наряду с функцией `memo`, часто используются чрезмерно. React создан быть быстрым. Он разработан для того, чтобы отображать много компонентов. Процесс оптимизации производительности начался уже тогда, когда вы впервые решили использовать React. Он уже быстрый. Любой дальнейший рефакторинг может быть только крайней мерой.

Рефакторинг влечет за собой компромиссы. Частое использование хуков `useCallback` и `useMemo` может снизить производительность приложения, которое требует больше строк кода и больше времени разработчика. Когда вы проводите рефакторинг для повышения производительности, у него должна быть цель. Возможно, вы хотите, чтобы экран не зависал и не мерцал. Возможно, некоторые трудоемкие функции необоснованно замедляют скорость приложения.

Для измерения производительности компонентов можно использовать React Profiler. Он входит в набор React Development Tools, который вы, вероятно, уже установили (доступен для Chrome (oreil.ly/1UNct) и Firefox (oreil.ly/0NYbR)).

Перед рефакторингом всегда проверяйте, работает ли приложение и устраивает ли вас кодовая база. Чрезмерный рефакторинг или его проведение до того, как приложение заработает, может привести к появлению странных ошибок, которые трудно обнаружить, и вам, возможно, не стоит тратить не него время и лучше заняться оптимизацией.

В предыдущих двух главах мы рассмотрели встроенные в React хуки. Вы увидели варианты использования каждого хука и создавали свои хуки. Далее мы будем развивать эти базовые навыки, добавляя дополнительные библиотеки и продвинутые паттерны.

ГЛАВА 8

Включение данных

Данные — это кровь, которая течет по всему приложению и питает его компоненты. Составленные нами компоненты пользовательского интерфейса — это сосуды для данных. Мы наполняем приложения данными из интернета. Собираем, создаем и отправляем новые данные в интернет. Ценность приложений не в компонентах, а в данных, которые проходят через компоненты.

Поэтому данные также можно сравнить с водой или едой. *Облако* — это бесконечный источник, из которого мы отправляем и получаем данные. Это интернет. Это сети, службы, системы и базы данных, в которых мы перемещаем и храним зеттабайты данных. Облако *снабжает* наших клиентов самыми свежими данными из источника. Мы работаем с этими данными локально и даже храним их локально. Но когда наши локальные данные перестают синхронизироваться с источником, они теряют актуальность и считаются *устаревшими*.

Задача, известная всем разработчикам, использующим данные, — приложения должны получать свежие данные из облака. В этой главе мы рассмотрим различные методы загрузки данных из источника и работы с ними.

Запрос данных

В фильме «Звездные войны» C-3PO был протокольным дроидом. Его специальностью было, конечно же, общение. Он говорил более чем на шести миллионах языков. И, разумеется, C-3P0 умел отправлять HTTP-запросы, потому что протокол передачи гипертекста — один из самых популярных способов передачи данных через интернет.

HTTP — это основа интернет-общения. Каждый раз, заходя на главную страницу Google (http://www.google.com), мы просим сервер Google отправить нам форму поиска. Файлы, необходимые для поиска, передаются в браузер по протоколу HTTP.

Когда мы пишем в Google запрос «фотки котиков смотреть онлайн», Google отвечает данными, и изображения передаются в наш браузер по HTTP.

В JavaScript самый популярный способ сделать HTTP-запрос — использовать функцию `fetch` (выборку). Чтобы запросить у GitHub информацию о пользователе MoonHighway, отправим запрос на выборку:

```
fetch(`https://api.github.com/users/moonhighway`)
  .then(response => response.json())
  .then(console.log)
  .catch(console.error);
```

Функция `fetch` возвращает промис. Мы делаем асинхронный запрос к определенному URL: https://api.github.com/users/moonhighway. Требуется время, чтобы он прошел через интернет и передал информацию в обратный вызов с помощью метода `.then(callback)`. API GitHub отвечает данными JSON, но эти данные содержатся в теле запроса HTTP, поэтому мы вызываем метод `response.json()`, чтобы получить эти данные и проанализировать их как JSON. После получения данных мы записываем их в консоль. Если что-то пойдет не так, мы передадим ошибку в метод `console.error`.

GitHub ответит на этот запрос объектом JSON:

```
{
  "login": "MoonHighway",
  "id": 5952087,
  "node_id": "MDEyOk9yZ2FuaXphdGlvbjU5NTIwODc=",
  "avatar_url": "https://avatars0.githubusercontent.com/u/5952087?v=4",
  "bio": "Web Development classroom training materials.",

  ...

}
```

На GitHub основная информация об учетных записях пользователей предоставляется через их API. Например, попробуйте поискать себя: https://api.github.com/users/<YOUR_GIT- HUB_USER_NAME>.

Другой способ работы с промисами — использовать `async` и `await`. Поскольку `fetch` возвращает промис, мы можем ожидать запрос на выборку внутри функции `async`:

```
async function requestGithubUser(githubLogin) {
  try {
    const response = await fetch(
      `https://api.github.com/users/${githubLogin}`
    );
    const userData = await response.json();
    console.log(userData);
```

```
  } catch (error) {
    console.error(error);
  }
}
```

Этот код дает те же результаты, что и предыдущий запрос `fetch`, выполненный путем связывания функций `.then` с запросом. Когда мы ожидаем промис, следующая строка кода не выполнится, пока промис не будет выполнен. Это удобный способ работы с промисами в коде. Мы будем использовать оба эти подхода до конца этой главы.

Отправка данных с запросом

Многие запросы требуют загрузки данных. Например, соберем информацию о пользователе, чтобы создать учетную запись, или найдем новую информацию о пользователе для обновления его учетной записи.

Обычно мы используем POST-запрос при создании данных и PUT-запрос, когда мы их изменяем. Второй аргумент функции `fetch` позволяет передать объект с параметрами, которые `fetch` может использовать при создании HTTP-запроса:

```
fetch("/create/user", {
  method: "POST",
  body: JSON.stringify({ username, password, bio })
});
```

В этой функции для создания нового пользователя используется метод POST. Имя пользователя, пароль и биография пользователя передаются в виде строкового содержимого в теле запроса.

Загрузка файлов с помощью функции fetch

Для загрузки файлов требуется другой тип HTTP-запроса: запрос `multipart-formdata`. Этот тип запроса сообщает серверу, что в теле запроса находится один или несколько файлов. Чтобы сделать этот запрос в JavaScript, передадим объект `FormData` в теле запроса:

```
const formData = new FormData();
formData.append("username", "moontahoe");
formData.append("fullname", "Alex Banks");
forData.append("avatar", imgFile);

fetch("/create/user", {
  method: "POST",
  body: formData
});
```

На этот раз, когда мы создаем пользователя, мы передаем имя пользователя, полное имя и аватар вместе с запросом в виде объекта `formData`. Хотя эти значения здесь жестко запрограммированы, мы могли бы легко получить их из формы.

Авторизованные запросы

Иногда для отправки запросов нужно авторизоваться. Обычно авторизация требуется для получения личных или конфиденциальных данных. Кроме того, пользователям авторизация почти всегда требуется для выполнения действий на сервере с помощью запросов POST, PUT или DELETE.

Пользователи обычно идентифицируют себя с каждым запросом, добавляя к запросу уникальный токен, который служба может использовать для идентификации пользователя. Этот токен обычно передается как заголовок `Authorization`. На GitHub вы можете увидеть информацию о своей личной учетной записи, если отправите токен вместе со своим запросом:

```
fetch(`https://api.github.com/users/${login}`, {
  method: "GET",
  headers: {
    Authorization: `Bearer ${token}`
  }
});
```

Обычно пользователь получает токены при входе в систему, указав свое имя пользователя и пароль. Также токены мы можем получить от третьих лиц, таких как GitHub или Facebook, используя открытый стандартный протокол OAuth.

GitHub позволяет создавать персональный токен пользователя. Вы можете создать его, войдя в GitHub и перейдя в меню Settings > Developer > Settings > Personal Access Tokens. Здесь вы можете создавать токены с определенными правилами чтения и записи, а затем использовать их для получения личной информации из GitHub API. Если вы создадите токен личного доступа и отправите его вместе с запросом на выборку, GitHub предоставит дополнительную личную информацию о вашей учетной записи.

Получение данных из компонента React требует наличия двух хуков: `useState` нужен для сохранения ответа в состоянии, а `useEffect` нужен для выполнения запроса выборки. Например, если мы хотим отобразить информацию о пользователе GitHub в компоненте, то можем использовать следующий код:

```
import React, { useState, useEffect } from "react";

function GitHubUser({ login }) {
  const [data, setData] = useState();
```

```
  useEffect(() => {
    if (!login) return;
    fetch(`https://api.github.com/users/${login}`)
      .then(response => response.json())
      .then(setData)
      .catch(console.error);
  }, [login]);

  if (data)
    return <pre>{JSON.stringify(data, null, 2)}</pre>;

  return null;
}

export default function App() {
  return <GitHubUser login="moonhighway" />;
}
```

Компонент `App` отображает компонент `GitHubUser` и данные о `moonhighway` в формате JSON. При первом рендеринге `GitHubUser` устанавливает переменную состояния для данных с помощью хука `useState`. Затем, поскольку данные изначально равны `null`, компонент возвращает `null`, что заставляет React ничего не отображать. Это не ошибка: мы просто увидим черный экран.

После рендеринга компонента вызывается хук `useEffect`. Здесь мы делаем запрос на выборку. Получив ответ, мы получаем и анализируем данные в этом ответе как JSON. Теперь мы можем передать этот объект JSON в функцию `setData`, которая заставит компонент отобразиться еще раз, но на этот раз у него будут данные. Хук `useEffect` не будет вызываться снова, пока не изменится значение `login`. Когда это произойдет, нам нужно будет запросить дополнительную информацию о другом пользователе из GitHub.

Имея данные в объекте `data`, мы отображаем их как строку JSON в элементе `pre`. Метод `JSON.stringify` принимает три аргумента: данные JSON для преобразования в строку; функцию-заменитель, которую можно использовать для замены свойств объекта JSON; количество пробелов, используемых при форматировании данных. Здесь заменитель — `null`, потому что мы не хотим ничего заменять. Число `2` — количество пробелов, которые будут использоваться при форматировании кода. То есть отступ строки JSON будет составлять два пробела. Использование элемента `pre` учитывает пробелы, поэтому в конечном итоге отображается читаемый JSON.

Сохранение данных локально

Мы можем сохранять данные локально в браузере через Web Storage API с помощью объектов `window.localStorage` или `window.sessionStorage`. `sessionStorage`

API сохраняет данные только для сеанса пользователя: при закрытии вкладок или перезапуске браузера эти данные удалятся. `localStoragt` будет хранить данные до тех пор, пока вы их не удалите.

Данные JSON хранятся в хранилище браузера в виде строки. Это подразумевает преобразование объекта в строку JSON перед его сохранением и анализ этой строки в JSON при ее загрузке. Функция для обработки сохранения и загрузки данных JSON в браузер может выглядеть так:

```
const loadJSON = key =>
  key && JSON.parse(localStorage.getItem(key));
const saveJSON = (key, data) =>
  localStorage.setItem(key, JSON.stringify(data));
```

Функция `loadJSON` загружает элемент из `localStorage` с помощью ключа. Локальная функция `Storage.getItem` используется для загрузки данных. Если элемент там есть, он анализируется в JSON перед возвратом. В противном случае функция `loadJSON` возвращает `null`.

Функция `saveJSON` сохраняет некоторые данные в `localStorage` с помощью уникального идентификатора `key`. Мы можем использовать функцию `localStorage.setItem` для сохранения данных в браузере, предварительно преобразовав их в строку JSON.

Загрузка данных из веб-хранилища, сохранение данных в веб-хранилище, преобразование данных в строки и анализ строк JSON — все это синхронные задачи. Функции `loadJSON` и `saveJSON` тоже синхронные. Будьте осторожны — слишком частый вызов этих функций со слишком большим объемом данных может привести к проблемам с производительностью. Поэтому рекомендуется ограничивать или блокировать эти функции.

Если мы сохраним в `localStorage` данные пользователя, полученные из нашего запроса GitHub, то при выполнении следующего запроса на того же пользователя мы сможем использовать их вместо отправки на GitHub другого запроса. Добавим в компонент `GitHubUser` следующий код:

```
const [data, setData] = useState(loadJSON(`user:${login}`));
useEffect(() => {
  if (!data) return;
  if (data.login === login) return;
  const { name, avatar_url, location } = data;
  saveJSON(`user:${login}`, {
    name,
    login,
    avatar_url,
    location
  });
}, [data]);
```

Функция `loadJSON` является синхронной, поэтому мы можем использовать ее при вызове `useState` для установки начального значения для данных. Если в браузере сохранены пользовательские данные под именем `user:moonhighway`, мы установим данные, используя это значение. В противном случае данные изначально будут иметь значение `null`.

Когда данные изменяются после загрузки из GitHub, мы вызываем `saveJSON`, чтобы сохранить только те данные пользователя, которые нам нужны: `name`, `login`, `avatar_url` и `location`. Нет необходимости сохранять остальную часть пользовательского объекта, если мы его не используем. Мы также пропускаем сохранение данных `!data`, когда этот объект пуст. Кроме того, если текущий логин и `data.login` равны, значит, мы уже сохранили данные для этого пользователя.

Посмотрим на компонент `GitHubUser`, который использует `localStorage` для сохранения данных в браузере:

```
import React, { useState, useEffect } from "react";

const loadJSON = key =>
  key && JSON.parse(localStorage.getItem(key));
const saveJSON = (key, data) =>
  localStorage.setItem(key, JSON.stringify(data));

function GitHubUser({ login }) {
  const [data, setData] = useState(
    loadJSON(`user:${login}`)
  );

  useEffect(() => {
    if (!data) return;
    if (data.login === login) return;
    const { name, avatar_url, location } = data;
    saveJSON(`user:${login}`, {
      name,
      login,
      avatar_url,
      location
    });
  }, [data]);

  useEffect(() => {
    if (!login) return;
    if (data && data.login === login) return;
    fetch(`https://api.github.com/users/${login}`)
      .then(response => response.json())
      .then(setData)
      .catch(console.error);
```

```
  }, [login]);

  if (data)
    return <pre>{JSON.stringify(data, null, 2)}</pre>;

  return null;
}
```

Обратите внимание, что у компонента `GitHubUser` теперь есть два хука `useEffect`. Первый хук используется для сохранения данных в браузере. Он вызывается всякий раз, когда изменяется значение данных. Второй хук используется для запроса дополнительных данных из GitHub. Запрос на выборку не отправляется, если данные этого пользователя ранее были сохранены локально. Они обрабатываются вторым оператором `if` во втором хуке: `if (data && data.login === login) return;`. Если данные есть и логин для них совпадает со свойством логина, тогда нет необходимости отправлять дополнительный запрос в GitHub. Мы просто будем использовать локальные данные.

При первом запуске приложения, если для входа в систему задано значение `moonhighway`, на странице будет отображаться следующий объект:

```
{
  "login": "MoonHighway",
  "id": 5952087,
  "node_id": "MDEyOk9yZ2FuaXphdGlvbjU5NTIwODc=",
  "avatar_url": "https://avatars0.githubusercontent.com/u/5952087?v=4",
  "gravatar_id": "",
  "url": "https://api.github.com/users/MoonHighway",
  "html_url": "https://github.com/MoonHighway",

  ...

}
```

Перед нами ответ GitHub. Это ясно видно, поскольку этот объект содержит много дополнительной информации о пользователе, которая нам не нужна. При первом запуске этой страницы мы увидим такой длинный ответ. Но при втором запуске страницы ответ будет намного короче:

```
{
  "name": "Moon Highway",
  "login": "moonhighway",
  "avatar_url": "https://avatars0.githubusercontent.com/u/5952087?v=4",
  "location": "Tahoe City, CA"
}
```

На этот раз в браузере отображаются данные `moonhighway`, которые мы сохранили локально. Поскольку нам нужно четыре поля данных, мы сохранили только

их. Пока мы не очистим хранилище, на запрос будет отображаться именно этот сокращенный ответ:

```
localStorage.clear();
```

И `sessionStorage`, и `localStorage` — важное оружие веб-разработчиков. Мы можем работать с локальными данными без подключения к сети, и они позволят повысить производительность приложений за счет сокращения числа сетевых запросов. Однако мы должны знать, когда их использовать. Внедрение автономного хранилища усложнит приложение и может затруднить его разработку. Кроме того, нам не нужно работать с веб-хранилищем для кеширования данных. Если мы просто ищем повышения производительности, то можем попробовать разрешить HTTP обрабатывать кеширование. Браузер будет автоматически кэшировать контент, если мы добавим в заголовки строку `Cache-Control`, поскольку `max-age = <EXP_DATE>`. `EXP_DATE` определит дату истечения срока действия контента.

Обработка состояний промисов

HTTP-запросы и промисы имеют три состояния: ожидание, успех (выполнен) и сбой (отклонен). Перед получением результата запрос находится *в ожидании*. Результат может быть одним из двух: успех или неудача. Успешное подключение к серверу и получение данных означает, что промис *выполнен*. Если что-то пошло не так, мы можем сказать, что HTTP-запрос не прошел или промис *отклонен*. В обоих случаях мы получим сообщение об ошибке, где будет изложено, что произошло.

Все три состояния нужно обрабатывать. Изменим пользовательский компонент GitHub, чтобы он отображал не только успешный ответ. И добавим сообщение «loading...» для периода ожидания, а также данные ошибки, если что-то пойдет не так:

```
function GitHubUser({ login }) {
  const [data, setData] = useState();
  const [error, setError] = useState();
  const [loading, setLoading] = useState(false);

  useEffect(() => {
    if (!login) return;
    setLoading(true);
    fetch(`https://api.github.com/users/${login}`)
      .then(data => data.json())
      .then(setData)
      .then(() => setLoading(false))
      .catch(setError);
```

```
  }, [login]);

  if (loading) return <h1>loading...</h1>;
  if (error)
    return <pre>{JSON.stringify(error, null, 2)}</pre>;
  if (!data) return null;

  return (
    <div className="githubUser">
      <img
        src={data.avatar_url}
        alt={data.login}
        style={{ width: 200 }}
      />
      <div>
        <h1>{data.login}</h1>
        {data.name && <p>{data.name}</p>}
        {data.location && <p>{data.location}</p>}
      </div>
    </div>
  );
}
```

Когда запрос выполнен успешно, информация о MoonHighway появится на экране (рис. 8.1).

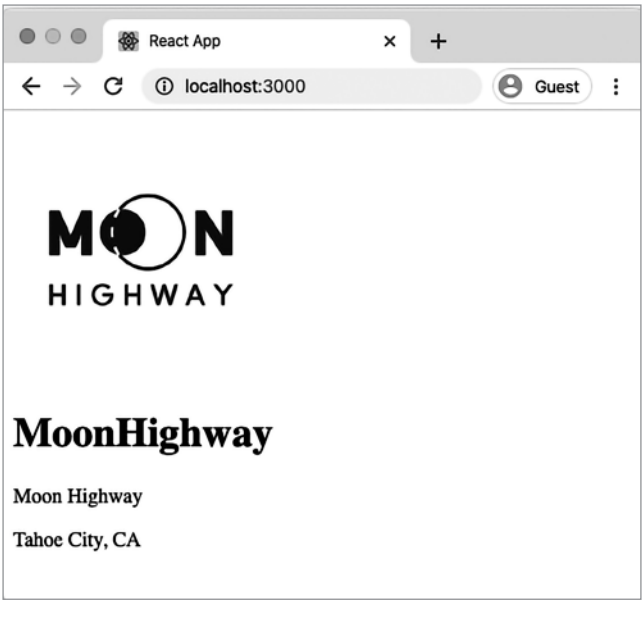


Рис. 8.1. Пример вывода

Если что-то пойдет не так, мы просто отобразим объект ошибки в виде строки JSON. В продакшене мы бы поработали с ошибкой больше. Может быть, отследили бы ее, записали в журнал или попробовали сделать еще один запрос. Но во время разработки можно отображать сведения об ошибках — это дает разработчику мгновенную информацию о происходящем.

Наконец, пока запрос находится в ожидании, мы просто отображаем сообщение «loading...» с использованием `h1`.

Иногда HTTP-запрос может завершиться ошибкой. Это происходит, когда запрос сам по себе выполнился успешно — то есть подключился к серверу и получил ответ, но тело ответа содержит ошибку. Иногда серверы передают дополнительные ошибки как успешные ответы.

Обработка всех трех состояний немного увеличивает объем кода, но ее необходимо проводить при каждом запросе. Запросы требуют времени, и многое в них может пойти не так. Поскольку все запросы и промисы имеют эти три состояния, это позволяет обрабатывать все HTTP-запросы с помощью многоразового хука, компонента или даже функции `Suspense`. Мы рассмотрим каждый из этих подходов, но сначала определим концепцию рендер-пропсов.

Рендер-пропсы

Термин *рендер-пропс* (render props) в переводе означает отображение свойств. Это могут быть компоненты, отправленные как свойства, которые отображаются при соблюдении определенных условий, или свойства функций, которые возвращают отображаемые компоненты. Во втором случае данные-функции могут передаваться в качестве аргументов и использоваться при рендеринге возвращаемого компонента.

Рендер-пропсы полезны при максимальном увеличении возможности повторного использования в асинхронных компонентах. С помощью этого паттерна мы можем создавать компоненты, которые абстрагируются от сложной механики или монотонного паттерна разработки приложений.

Рассмотрим задачу рендеринга списка:

```
import React from "react";

const tahoe_peaks = [
  { name: "Freel Peak", elevation: 10891 },
  { name: "Monument Peak", elevation: 10067 },
  { name: "Pyramid Peak", elevation: 9983 },
  { name: "Mt. Tallac", elevation: 9735 }
];
```

```
export default function App() {
  return (
    <ul>
      {tahoe_peaks.map((peak, i) => (
        <li key={i}>
          {peak.name} — {peak.elevation.toLocaleString()}ft
        </li>
      ))}
    </ul>
  );
}
```

В этом примере четыре высочайших вершины в Тахо выводятся в неупорядоченном списке. Код понятный, но рендеринг каждого элемента массива по отдельности вносит некоторую сложность в код. Рендеринг массива элементов является довольно распространенной задачей. Создадим компонент `List`, который можно будет повторно использовать для отображения неупорядоченного списка.

В JavaScript массивы либо содержат значения, либо являются пустыми. Когда список пуст, нужно сообщить об этом пользователю. Однако это сообщение может измениться после реализации. Но не беспокойтесь: мы можем передать компонент для рендеринга, когда список пуст:

```
function List({ data = [], renderEmpty }) {
  if (!data.length) return renderEmpty;
  return <p>{data.length} items</p>;
}

export default function App() {
  return <List renderEmpty={<p>This list is empty</p>} />;
}
```

Компонент `List` ожидает два свойства: `data` и `renderEmpty`. Первый аргумент, `data`, представляет собой массив элементов, которые нужно отобразить. Его значение по умолчанию — пустой массив. Второй аргумент `renderEmpty` — это компонент, который будет отображаться, если список пуст. Поэтому, когда `data.length` равна `0`, компонент `List` отображает все, что было передано как свойство `renderEmpty`, возвращая это свойство.

В этом случае пользователи увидят сообщение «This list is empty».

`renderEmpty` — это рендер-пропс, потому что он содержит компонент для рендеринга при выполнении определенного условия — в данном случае когда список пуст или свойство данных не предоставлено.

Мы можем передать этому компоненту реальный массив данных:

```
export default function App() {
  return (
    <List
      data={tahoe_peaks}
      renderEmpty={<p>This list is empty</p>}
    />
  );
}
```

На этом этапе отображается только количество элементов, имеющихся в массиве: четыре.

Мы также можем указать компоненту `List`, что отображать для каждого элемента, найденного в массиве. Например, мы можем отправить свойство `renderItem`:

```
export default function App() {
  return (
    <List
      data={tahoe_peaks}
      renderEmpty={<p>This list is empty</p>}
      renderItem={item => (
        <>
          {item.name} – {item.elevation.toLocaleString()}ft
        </>
      )}
    />
  );
}
```

На этот раз рендер-пропс — это функция. Данные (сам элемент) передаются в эту функцию в качестве аргумента, так что ее можно использовать для отображения каждой вершины. В этом случае мы отображаем фрагмент React, в котором содержится имя и высота элемента. Для массива `tahoe_peaks` мы ожидаем, что свойство `renderItem` будет вызываться четыре раза: по одному на каждую вершину в массиве.

Этот подход позволяет абстрагироваться от механики рендеринга массивов. Теперь компонент `List` будет выполнять все сам, а нам останется указать, что вывести на экран:

```
function List({ data = [], renderItem, renderEmpty }) {
  return !data.length ? (
    renderEmpty
  ) : (
    <ul>
```

```
      {data.map((item, i) => (
        <li key={i}>{renderItem(item)}</li>
      ))}
    </ul>
  );
}
```

Когда массив данных не пуст, компонент `List` отображает неупорядоченный список `<ul>`. Он отображает каждый элемент в массиве с помощью метода `.map` и элементы списка `<li>` для каждого значения в массиве. Компонент `List` гарантирует, что каждый элемент списка получает уникальный ключ. В каждом элементе `<li>` вызывается свойство `renderItem`, а сам элемент передается этому свойству функции в качестве аргумента. Результатом является неупорядоченный список, в котором отображаются названия и высота каждой из самых высоких вершин Тахо.

Хорошей новостью является то, что у нас есть многоразовый компонент `List`, который мы можем использовать всякий раз, когда нам нужно отобразить неупорядоченный список. Плохая новость в том, что наш компонент недостаточно хорош. Есть более совершенные компоненты для решения этой задачи.

Виртуализированные списки

Чтобы разработать многоразовый компонент для рендеринга списков, необходимо рассмотреть множество различных вариантов использования и способов реализации такого компонента. Одна из самых важных вещей, которую следует учитывать, — это случай, когда список очень большой. Объемы данных, с которыми мы работаем в продакшене, кажутся бесконечными. Поиск Google выполняет выборку страниц результатов. Поиск жилья в Тахо на Airbnb выводит бесконечный список домов и квартир.

Возможности браузера с точки зрения рендеринга ограничены. Рендеринг требует времени, вычислительной мощности и памяти, и все эти ресурсы исчерпаемы. Это следует учитывать при разработке многоразового компонента списка. Что нам делать, когда массив данных очень велик?

Несмотря на то что поиск места для проживания может дать тысячу результатов, мы не можем просмотреть их все одновременно, так как на экране попросту недостаточно места для всех изображений, названий и цен. Мы видим только пять результатов за раз. Прокручивая страницу, мы можем увидеть больше результатов, но для вывода всей тысячи крутить придется долго. Рендеринг тысячи результатов в прокручиваемом слое потребует значительных ресурсов смартфона.

Что, если бы мы отображали только 11 результатов, а не 1000 за раз? Помните, что пользователь все равно может видеть на одном экране около пяти результатов. Таким образом, мы отображаем пять элементов, которые может видеть пользователь, и еще шесть элементов за пределами экрана как над, так и под видимой областью. Рендеринг элементов над и под видимым окном позволит пользователю выполнять прокрутку в обоих направлениях (рис. 8.2).

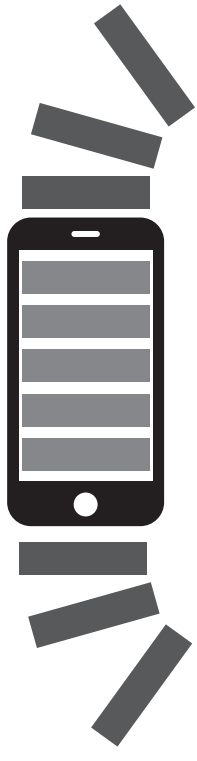


Рис. 8.2. Окно с контентом за пределами экрана

По мере того как пользователь прокручивает страницу, мы можем убирать из рендеринга результаты, которые уже были просмотрены, одновременно с этим отображая новые результаты за пределами экрана, готовя их для будущей прокрутки. Такое решение означает, что браузер всегда будет единовременно отображать только 11 элементов, а данные для остальных элементов будут ждать своей очереди. Этот метод называется *кадрированием* или *виртуализацией*. Он позволяет прокручивать очень большие, иногда бесконечные списки данных без сбоев браузера.

При создании компонента виртуализированного списка нужно многое учитывать. К счастью, нам не придется начинать с нуля, так как сообщество уже разработало множество готовых компонентов. Самые популярные из них для браузера — это react-window и react-virtualized. Виртуализированные списки настолько важны, что в React Native есть встроенный компонент `FlatList`. Большинству из нас не придется создавать компоненты виртуализированных списков, но нужно знать, как их использовать.

Чтобы реализовать виртуализированный список, нам понадобится много данных — пока фиктивных:

```
npm I faker
```

Установка `faker` позволит нам создать большой массив фиктивных данных. Создадим случайным образом пять тысяч фиктивных пользователей:

```
import faker from "faker";

const bigList = [...Array(5000)].map(() => ({
  name: faker.name.findName(),
  email: faker.internet.email(),
  avatar: faker.internet.avatar()
}));
```

Переменная `bigList` создана путем сопоставления массива из пяти тысяч пустых значений и их замены на информацию о пользователях. Имя, адрес электронной почты и аватар каждого пользователя генерируются случайным образом с помощью функций библиотеки `faker`.

Если мы используем компонент `List`, который создали в предыдущем разделе, он отобразит все пять тысяч пользователей одновременно:

```
export default function App() {
  const renderItem = item => (
    <div style={{ display: "flex" }}>
      <img src={item.avatar} alt={item.name} width={50} />
      <p>
        {item.name} – {item.email}
      </p>
    </div>
  );

  return <List data={bigList} renderItem={renderItem} />;
}
```

Этот код создает элемент `div` для каждого пользователя. Внутри `div` отображается элемент `img` — аватар пользователя, а имя пользователя и адрес электронной почты отображаются элементом `<p>` (рис. 8.3).

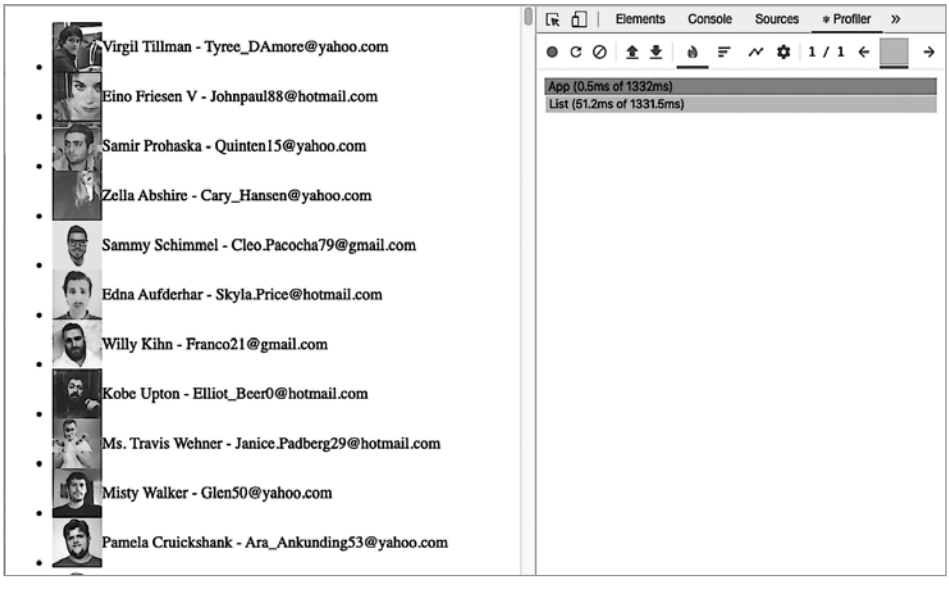


Рис. 8.3. Результаты работы кода

Работа React и современных браузеров впечатляет. Скорее всего, мы сумеем отобразить все пять тысяч пользователей, хотя это и займет некоторое время. В этом примере, если быть точным, потребовалось 52 мс. Вместе с числом пользователей в списке растет и длительность обработки, пока не настанет переломный момент.

Отобразим тот же список пользователей с помощью react-window:

```
npm i react-window
```

react-window — это библиотека, в которой есть несколько компонентов, позволяющих отображать виртуализированные списки. В этом примере мы используем компонент `FixSizeList` из react-window:

```
import React from "react";
import { FixedSizeList } from "react-window";
import faker from "faker";

const bigList = [...Array(5000)].map(() => ({
  name: faker.name.findName(),
  email: faker.internet.email(),
  avatar: faker.internet.avatar()
}));
export default function App() {
  const renderRow = ({ index, style }) => (
    <div style={{ ...style, ...{ display: "flex" } }}>
      <img
        src={bigList[index].avatar}
        alt={bigList[index].name}
        width={50}
      />
      <p>
        {bigList[index].name} – {bigList[index].email}
      </p>
    </div>
  );

  return (
    <FixedSizeList
      height={window.innerHeight}
      width={window.innerWidth – 20}
      itemCount={bigList.length}
      itemSize={50}
    >
      {renderRow}
    </FixedSizeList>
  );
}
```

Компонент `FixedSizeList` немного отличается от компонента `List`. Ему требуется общее количество элементов в списке и количество пикселов, которое требуется каждой строке, в качестве свойства `itemSize`. Еще одно большое различие в синтаксисе заключается в том, что рендер-проп передается в `FixedSizeList` как свойство `children`. Этот паттерн рендеринга свойств используется довольно часто.

Итак, посмотрим, что произойдет, если мы отобразим пять тысяч пользователей с помощью компонента `Fix SizeList` (рис. 8.4).

На этот раз не все пользователи вывелись одновременно. Отображаются только те строки, которые пользователю видны или находятся близко. Обратите внимание, что для этого начального рендеринга требуется всего 2,6 мс.

Когда вы прокручиваете вниз, чтобы просмотреть следующих пользователей, `FixedSizeList` усердно работает, выводя больше пользователей за пределы экрана, а также удаляя пользователей, которые уже ушли из вашего «поля зрения». Этот компонент автоматически выполняет прокрутку в обоих направлениях, при этом он может отображаться довольно часто, но рендеринг выполняется быстро. Также не имеет значения, сколько пользователей в массиве — `FixedSizeList` справится.

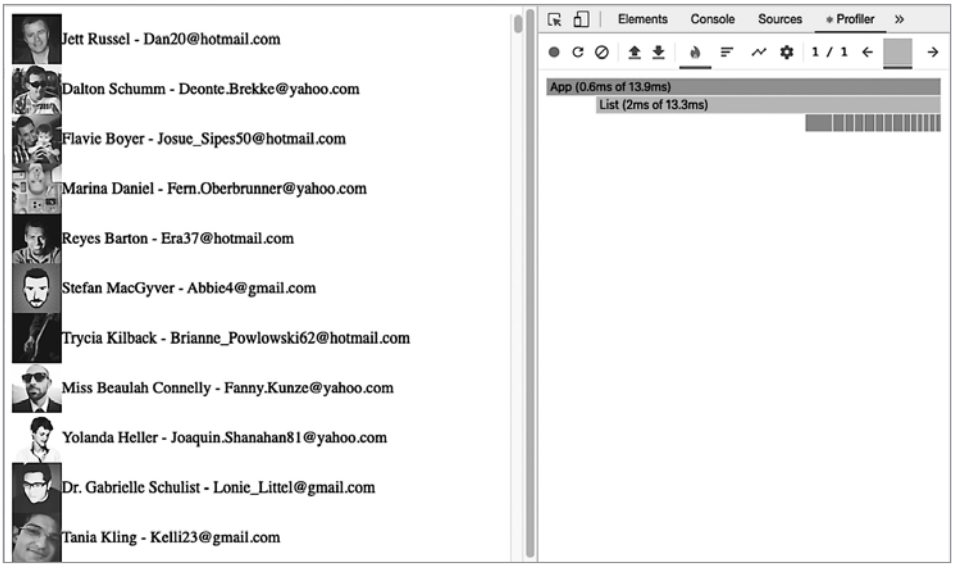


Рис. 8.4. Рендеринг занимает 2,6 мс

Создание хука Fetch

Мы знаем, что запрос может иметь три состояния: в ожидании, успех или неудача. Мы можем снова использовать логику, необходимую для выполнения запроса на выборку, создав пользовательский хук. Назовем его `useFetch` и будем использовать его в компонентах приложения всякий раз, когда нужно сделать запрос на выборку:

```
import React, { useState, useEffect } from "react";

export function useFetch(uri) {
  const [data, setData] = useState();
  const [error, setError] = useState();
  const [loading, setLoading] = useState(true);

  useEffect(() => {
    if (!uri) return;
    fetch(uri)
      .then(data => data.json())
      .then(setData)
      .then(() => setLoading(false))
      .catch(setError);
  }, [uri]);

  return {
    loading,
    data,
    error
  };
}
```

Этот хук стал результатом объединения хуков `useState` и `useEffect`. В нем представлены три состояния запроса на выборку: ожидание, успех и неудача. Когда запрос ожидает обработки, хук вернет значение `true` для переменной `loading`. Когда запрос выполняется успешно и данные получены, они будут переданы компоненту из этого хука. Если что-то пойдет не так, хук вернет ошибку.

Все три состояния управляются внутри хука `useEffect`. Этот хук вызывается каждый раз при изменении значения `uri`. Если `uri` отсутствует, запрос на выборку не выполняется. Когда `uri` есть, начинается запрос на выборку. Если запрос выполняется успешно, мы передаем полученный JSON в функцию `setData`, изменяя значение состояния для данных. После этого мы изменяем значение состояния `laoding` на `false`, потому что запрос выполнился (и, соответственно, больше не находится в ожидании). Если что-то пойдет не так, мы перехватим значение и передадим его в `setError`, который изменит значение состояния на `error`.

Теперь мы можем использовать этот хук для выполнения запросов на выборку в компонентах. Каждый раз, когда значения `loading`, `data` или `error` изменяются, хук заставляет компонент `GitHubUser` повторно отображать их новые значения:

```
function GitHubUser({ login }) {
  const { loading, data, error } = useFetch(
    `https://api.github.com/users/${login}`
  );

  if (loading) return <h1>loading...</h1>;
  if (error)
    return <pre>{JSON.stringify(error, null, 2)}</pre>;

  return (
    <div className="githubUser">
      <img
        src={data.avatar_url}
        alt={data.login}
        style={{ width: 200 }}
      />
      <div>
        <h1>{data.login}</h1>
        {data.name && <p>{data.name}</p>}
        {data.location && <p>{data.location}</p>}
      </div>
    </div>
  );
}
```

Хотя в компоненте теперь меньше логики, он по-прежнему обрабатывает все три состояния. Предполагая, что у нас есть компонент `SearchForm`, получающий значения из строки поиска от пользователя, мы можем добавить компонент `GitHubUser` к основному компоненту `App`:

```
import React, { useState } from "react";
import GitHubUser from "./GitHubUser";
import SearchForm from "./SearchForm";

export default function App() {
  const [login, setLogin] = useState("moontahoe");

  return (
    <>
      <SearchForm value={login} onSearch={setLogin} />
      <GitHubUser login={login} />
    </>
  );
}
```

Основной компонент App хранит имя пользователя GitHub в состоянии. Единственный способ изменить это значение — использовать форму поиска, чтобы найти нового пользователя. При изменении значения login значение, отправляемое в useFetch, тоже изменяется, поскольку оно зависит от свойства login: https://api.github.com/users/${login}. В результате меняется uri в хуке, и создается запрос на выборку нового пользователя. Мы создали специальный хук и использовали его для успешного создания небольшого приложения для поиска и рендеринга сведений о пользователях GitHub. Мы продолжим использовать этот хук при изменении приложения.

Создание компонента Fetch

Хуки обычно позволяют нам многократно использовать функциональность компонентов. Зачастую, когда дело доходит до рендеринга внутри компонентов мы используем один и тот же паттерн. Например, анимация загрузки, которую мы выбираем, может использоваться и в других местах приложения всякий раз, когда ожидается запрос на выборку. Способ обработки ошибок в запросах на выборку также может быть одинаковым во всем приложении.

Вместо того чтобы воспроизводить один и тот же код в нескольких компонентах, мы можем создать один компонент, который отображает анимацию загрузки и последовательно обрабатывает ошибки в выбранной области. Создадим компонент Fetch:

```
function Fetch({
  uri,
  renderSuccess,
  loadingFallback = <p>loading...</p>,
  renderError = error => (
    <pre>{JSON.stringify(error, null, 2)}</pre>
  )
}) {
  const { loading, data, error } = useFetch(uri);
  if (loading) return loadingFallback;
  if (error) return renderError(error);
  if (data) return renderSuccess({ data });
}
```

Пользовательский хук useFetch реализует один слой абстракции: он абстрагирует механику выполнения запроса на выборку. Компонент Fetch — это еще один уровень абстракции: он абстрагирует механику обработки того, что нужно отображать. Когда загружается запрос, компонент Fetch отображает все, что было передано в необязательное свойство loadFallback. В случае успеха данные ответа JSON передаются свойству renderSuccess. Если возникает ошибка, она отображается с помощью необязательного свойства renderError. Свойства

`loadingFallback` и `renderError` — это дополнительный уровень настройки. Но если их не задать, они вернутся к своим значениям по умолчанию.

Имея в своем арсенале компонент `Fetch`, мы можем упростить логику компонента `GitHubUser`:

```
import React from "react";
import Fetch from "./Fetch";

export default function GitHubUser({ login }) {
  return (
    <Fetch
      uri={`https://api.github.com/users/${login}`}
      renderSuccess={UserDetails}
    />
  );
}

function UserDetails({ data }) {
  return (
    <div className="githubUser">
      <img
        src={data.avatar_url}
        alt={data.login}
        style={{ width: 200 }}
      />
      <div>
        <h1>{data.login}</h1>
        {data.name && <p>{data.name}</p>}
        {data.location && <p>{data.location}</p>}
      </div>
    </div>
  );
}
```

Компонент `GitHubUser` получает логин, чтобы пользователь мог найти его на GitHub. Мы используем этот логин для создания свойства `uri`, которое отправляем компоненту `fetch`. В случае успеха визуализируется компонент `UserDetails`. Когда загружается компонент `Fetch`, будет отображаться сообщение по умолчанию «loading...» Если что-то пойдет не так, автоматически отобразятся сведения об ошибке.

Мы можем задать для этих свойств индивидуальные значения. Ниже приведен пример того, как мы можем альтернативно использовать наш гибкий компонент:

```
<Fetch
  uri={`https://api.github.com/users/${login}`}
  loadingFallback={<LoadingSpinner />}
  renderError={error => {
    // handle error
    return <p>Something went wrong... {error.message}</p>;
```

```
  }}
  renderSuccess={({ data }) => (
    <>
      <h1>Todo: Render UI for data</h1>
      <pre>{JSON.stringify(data, null, 2)}</pre>
    </>
  )}
/>
```

На этот раз компонент `Fetch` отобразит настраиваемый счетчик загрузки. В случае неудачи мы скроем детали ошибки. Если запрос будет выполнен успешно, мы отобразим необработанные данные вместе с сообщением `TODO` для себя.

Будьте осторожны: дополнительные уровни абстракции, будь то хуки или компоненты, могут усложнить код. Наша работа — снизить сложность везде, где это возможно. Сейчас мы снизили сложность, абстрагируя логику, многократно используемую в компоненте и хуке.

Обработка множественных запросов

Начав запрашивать данные из интернета, мы уже не можем остановиться. Чаще всего нам приходится делать несколько HTTP-запросов, чтобы получить все данные, необходимые для работы приложения. Например, сейчас мы просим GitHub предоставить информацию об учетной записи пользователя. Нам также нужно получить информацию о репозиториях этого пользователя. Обе эти точки данных получаются путем выполнения отдельных HTTP-запросов.

У пользователей GitHub обычно много репозиториев. Информация о репозиториях пользователя передается в виде массива объектов. Создадим настраиваемый хук `useIterator`, который позволит перебирать любой массив объектов:

```
export const useIterator = (
  items = [],
  initialIndex = 0
) => {
  const [i, setIndex] = useState(initialIndex);

  const prev = () => {
    if (i === 0) return setIndex(items.length - 1);
    setIndex(i - 1);
  };

  const next = () => {
    if (i === items.length - 1) return setIndex(0);
    setIndex(i + 1);
  };

  return [items[i], prev, next];
};
```

Этот хук позволит нам циклически перебирать любой массив. Поскольку он возвращает элементы внутри массива, деструктурируем массив, чтобы дать этим значениям осмысленные имена.

```
const [letter, previous, next] = useIterator([
  "a",
  "b",
  "c"
]);
```

В данном случае начальное значение `letter` — `a`. Если пользователь вызовет следующее знаечение, компонент выполнит рендеринг снова, но на этот раз значение буквы будет `b`. Вызовите `next` еще два раза, и значение `letter` снова станет `a`, потому что этот итератор возвращается к первому элементу в массиве, не позволяя индексу выйти за пределы массива.

Хук `useIterator` принимает массив `items` и начальный индекс. Ключевым значением хука является индекс `i`, созданный с помощью хука `useState`. `i` используется для идентификации текущего элемента в массиве. Этот хук возвращает текущий элемент, `item[i]` и функции для итерации по массиву: `prev` и `next`. Эти функции либо уменьшают, либо увеличивают значение `i`, вызывая `setIndex` и повторный рендеринг хука с новым индексом.

Запоминание значений

Хук `useIterator` весьма хорош. Но мы можем сделать его еще лучше, запомнив значение элемента, а также функцию для `prev` и `next`:

```
import React, { useCallback, useMemo } from "react";

export const useIterator = (
  items = [],
  initialValue = 0
) => {
  const [i, setIndex] = useState(initialValue);

  const prev = useCallback(() => {
    if (i === 0) return setIndex(items.length – 1);
    setIndex(i – 1);
  }, [i]);

  const next = useCallback(() => {
    if (i === items.length – 1) return setIndex(0);
    setIndex(i + 1);
  }, [i]);

  const item = useMemo(() => items[i], [i]);

  return [item || items[0], prev, next];
};
```

Здесь и `prev`, и `next` создаются с помощью хука `useCallback`. Это гарантирует, что функция для `prev` всегда будет одинаковой, пока значение для `i` не изменится. Точно так же значение `item` всегда будет указывать на один и тот же объект элемента, если значение `i` не изменится.

Запоминание этих значений не дает большого прироста производительности. Во всяком случае, прирост этот недостаточно велик, чтобы оправдать сложность кода. Однако когда потребитель использует компонент `useIterator` запомненные значения всегда будут указывать на один и тот же объект и функцию. Это упростит сравнение этих значений и их использование в массивах зависимостей.

Теперь создадим компонент меню репозитория. В нем мы используем хук `useIterator`, чтобы позволить пользователям циклически перемещаться по списку репозиториев:

```
< learning-react >
```

Нажав кнопку Next, вы увидите имя следующего репозитория, а нажав кнопку Back — увидите имя предыдущего репозитория. Создадим компонент `RepoMenu` для реализации этой функции:

```
import React from "react";
import { useIterator } from "../hooks";

export function RepoMenu({
  repositories,
  onSelect = f => f
}) {
  const [{ name }, previous, next] = useIterator(
    repositories
  );

  useEffect(() => {
    if (!name) return;
    onSelect(name);
  }, [name]);

  return (
    <div style={{ display: "flex" }}>
      <button onClick={previous}>&lt;</button>
      <p>{name}</p>
      <button onClick={next}>&gt;</button>
    </div>
  );
}
```

`RepoMenu` получает список `repositories` через свойство. Затем он деструктурирует `name` из текущего объекта репозитория, а также предыдущие и следующие функции из `useItera tor.&lt`. Объект `&lt` означает «меньше» и отображается

знаком <. Аналогично работает &gt — «больше». Это индикаторы для предыдущего и следующего значений. Когда пользователь нажимает на любой из них, компонент повторно отображается с новым именем репозитория. Если имя меняется, значит пользователь выбрал другой репозиторий, поэтому мы вызываем функцию onSelect и передаем имя нового репозитория этой функции в качестве аргумента.

Помните, что деструктуризация массива позволяет называть элементы как угодно. Несмотря на то что внутри хука мы назвали эти функции prev и next, в момент вызова хука можно изменить их на previous и next.

Теперь мы можем создать компонент UserRepositories. Он должен сначала запросить список репозиториев пользователя GitHub, получить его и передать компоненту RepoMenu:

```
import React from "react";
import Fetch from "./Fetch";
import RepoMenu from "./RepoMenu";

export default function UserRepositories({
  login,
  selectedRepo,
  onSelect = f => f
}) {
  return (
    <Fetch
      uri={`https://api.github.com/users/${login}/repos`}
      renderSuccess={({ data }) => (
        <RepoMenu
          repositories={data}
          selectedRepo={selectedRepo}
          onSelect={onSelect}
        />
      )}
    />
  );
}
```

Компоненту UserRepositories требуется значение login, по которому он выполнит запрос на выборку списка репозиториев. login используется для создания URI и передачи его компоненту Fetch. После успешного разрешения выборки мы отобразим Repo Menu вместе со списком репозиториев, который был возвращен из компонента Fetch в массиве data. Когда пользователь выберет другой репозиторий, мы просто передадим имя этого нового репозитория родительскому объекту:

```
function UserDetails({ data }) {
  return (
    <div className="githubUser">
```

```
      <img src={data.avatar_url} alt={data.login} style={{ width: 200 }} />
      <div>
        <h1>{data.login}</h1>
        {data.name && <p>{data.name}</p>}
        {data.location && <p>{data.location}</p>}
      </div>
      <UserRepositories
        login={data.login}
        onSelect={repoName => console.log(`${repoName} selected`)}
      />
  </div>
);
```

Теперь нам нужно добавить новый компонент в компонент `UserDetails`. Когда компонент `UserDetails` отображается, мы также визуализируем список репозиториев этого пользователя. Если `login` имеет значение `eveporcello`, визуализированный вывод для вышеуказанного компонента будет выглядеть примерно как на рис. 8.5.

Рис. 8.5. Вывод репозитория

Чтобы получить информацию об учетной записи Евы и список ее репозиториев, нам нужно отправить два отдельных HTTP-запроса. Большую часть жизни разработчик React проводит за выполнением запросов на информацию и объединением полученной информации в красивые приложения с пользовательскими интерфейсами. Дважды запросить информацию — это только начало. В следую-

щем разделе мы будем делать еще больше запросов к GitHub и выводить файл README.md для выбранного репозитория.

Каскадные запросы

В предыдущем разделе мы сделали два HTTP-запроса. Сначала запросили данные пользователя, а затем, получив эти данные, сделали второй запрос репозиториев этого пользователя. Эти запросы выполнялись один за другим.

Первый запрос делается, когда мы изначально получаем данные пользователя:

```
<Fetch
  uri={`https://api.github.com/users/${login}`}
  renderSuccess={UserDetails}
/>
```

Как только у нас есть данные пользователя, отображается компонент `UserDetails`. Он, в свою очередь, отображает `UserRepositories`, который затем отправляет запрос на выборку для репозиториев этого пользователя:

```
<Fetch
  uri={`https://api.github.com/users/${login}/repos`}
  renderSuccess={({ data }) => (
    <RepoMenu repositories={data} onSelect={onSelect} />
  )}
/>
```

Мы называем эти запросы *каскадными*, потому что они выполняются один за другим и зависят друг от друга. Если в запросе данных о пользователе что-то сломается, запрос репозиториев этого пользователя не выполнится.

Добавим каскаду еще несколько слоев. Сначала мы запрашиваем информацию о пользователе, затем список его репозиториев, а затем, имея этот список репозиториев, мы делаем запрос на файл README.md первого репозитория. По мере того как пользователь просматривает список репозиториев, мы будем делать дополнительные запросы на README всех прочих репозиториев.

Файлы README репозитория записываются с использованием языка Markdown, который представляет собой текстовый формат, легко преобразуемый в HTML с помощью компонента `ReactMarkdown`. Для начала установим библиотеку react-markdown:

```
npm i react-markdown
```

Запрос содержимого файла README репозитория также выполняется через каскад запросов. Сначала запросим данные к README-маршруту репозитория:

https://api.github.com/repos/${login}/${repo}/readme. GitHub ответит на запрос данными о файле README репозитория, но содержимое этого файла не передаст. Он предоставляет нам объект `download_url`, который мы можем использовать для запроса содержимого файла README. Но чтобы получить контент Markdown, придется сделать дополнительный запрос. Оба эти запроса можно сделать в одной асинхронной функции:

```
const loadReadme = async (login, repo) => {
  const uri = `https://api.github.com/repos/${login}/${repo}/readme`;
  const { download_url } = await fetch(uri).then(res =>
    res.json()
  );
  const markdown = await fetch(download_url).then(res =>
    res.text()
  );

  console.log(`Markdown for ${repo}\n\n${markdown}`);
};
```

Чтобы найти файл README репозитория, понадобится логин владельца репозитория и имя `repository`. Эти значения используются для создания уникального URL-адреса: https://api.github.com/repos/moonhighway/learning-react/readme. Когда запрос будет выполнен успешно, мы деструктурируем `download_url` из его ответа. Теперь мы можем использовать это значение для загрузки содержимого README. Для этого достаточно извлечь `download_url`. Мы проанализируем этот текст как текст — `res.text()` — а не как JSON, потому что тело ответа написано на Markdown.

Имея Markdown, отрендерим его, обернув функцию `loadReadme` внутри компонента React:

```
import React, {
  useState,
  useEffect,
  useCallback
} from "react";
import ReactMarkdown from "react-markdown";

export default function RepositoryReadme({ repo, login }) {
  const [loading, setLoading] = useState(false);
  const [error, setError] = useState();
  const [markdown, setMarkdown] = useState("");

  const loadReadme = useCallback(async (login, repo) => {
    setLoading(true);
    const uri = `https://api.github.com/repos/${login}/${repo}/readme`;
    const { download_url } = await fetch(uri).then(res =>
      res.json()
```

```
    );
    const markdown = await fetch(download_url).then(res =>
      res.text()
    );
    setMarkdown(markdown);
    setLoading(false);
  }, []);

  useEffect(() => {
    if (!repo || !login) return;
    loadReadme(login, repo).catch(setError);
  }, [repo]);

  if (error)
    return <pre>{JSON.stringify(error, null, 2)}</pre>;
  if (loading) return <p>Loading...</p>;

  return <ReactMarkdown source={markdown} />;
}
```

Мы добавили к компоненту функцию `loadReadme` с помощью хука `useCallback`, чтобы запоминать функцию при первоначальном рендеринге компонента. Эта функция теперь изменяет состояние загрузки на `true` перед запросом на выборку и снова меняет его на `false` после запроса. Когда контент Markdown будет получен, он сохранится в состоянии с помощью функции `setMarkdown`.

Затем нам нужно вызвать функцию `loadReadme`, поэтому мы добавляем хук `useEffect` для загрузки файла README после первоначального рендеринга компонента. Если по какой-то причине свойства `repo` и `login` отсутствуют, файл README не будет загружен. Массив зависимостей в этом хуке содержит `[repo]`. Дело в том, что мы хотим загрузить еще один README, если значение для `repo` изменится. Если при загрузке README что-то пойдет не так, файл будет перехвачен и отправлен в функцию `setError`.

Обратите внимание, что мы должны обрабатывать те же три состояния рендеринга, что и для каждого запроса на выборку: ожидание, успех и неудача. Наконец, когда у нас есть успешный ответ, сам Markdown визуализируется с помощью компонента `ReactMarkdown`.

Нам осталось отобразить компонент `RepositoryReadme` внутри компонента `RepoMenu`. Когда пользователь перебирает репозитории с помощью компонента `RepoMenu`, файл README для каждого репозитория также загружается и отображается:

```
export function RepoMenu({ repositories, login }) {
  const [{ name }, previous, next] = useIterator(
    repositories
  );
```

```
  return (
    <>
      <div style={{ display: "flex" }}>
        <button onClick={previous}>&lt;</button>
        <p>{name}</p>
        <button onClick={next}>&gt;</button>
      </div>
      <RepositoryReadme login={login} repo={name} />
    </>
  );
}
```

Теперь приложение последовательно выполняет несколько запросов. Сначала их четыре: мы запрашиваем сведения о пользователе, список его репозиториев, информацию о README выбранного репозитория и, наконец, текстовое содержимое README. Это каскад запросов, потому что они выполняются один за другим.

Кроме того, когда пользователь взаимодействует с приложением, выполняется больше запросов. Для получения файла README каждый раз, когда пользователь меняет текущий репозиторий, выполняются два каскадных запроса. Все четыре начальных запроса выполняются каждый раз, когда пользователь ищет другую учетную запись GitHub.

Регулирование скорости передачи

Все вышеназванные запросы отображаются на вкладке Network в инструментах разработчика, благодаря чему вы сможете регулировать скорость передачи данных, чтобы посмотреть, как эти запросы будут работать в медленных сетях. Чтобы увидеть, как выполняется каскад запросов, нужно снизить скорость сети и отсматривать сообщения о загрузке по мере их отображения.

Вкладка Network доступна в инструментах разработчика большинства основных браузеров. Чтобы ограничить скорость сети в Google Chrome, нажмите стрелку рядом со словом Online (рис. 8.6).

Рис. 8.6. Изменение скорости сетевого запроса

Откроется меню, где вы сможете выбрать различные скорости (рис. 8.7).

Выбор Fast 3G или Slow 3G значительно ограничит сетевые запросы.

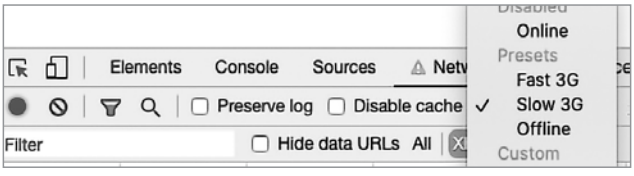


Рис. 8.7. Выбор скорости выполнения запроса

Кроме того, на вкладке Network отображается шкала времени для всех HTTP-запросов. Вы можете отфильтровать ее, чтобы просматривать только запросы «XHR», то есть выполненные с помощью `fetch` (рис. 8.8).

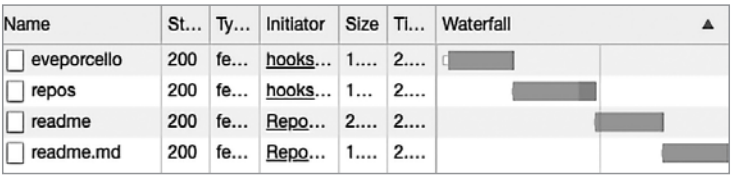


Рис. 8.8. Каскад запросов

На рисунке видно, что один за другим выполнилось четыре запроса. Обратите внимание, что изображение процесса загрузки называется Waterfall (водопад) из-за визуального сходства каскада запросов с водопадом. Это показывает, что каждый запрос выполняется после завершения другого.

Параллельные запросы

Иногда можно сделать приложение быстрее, отправив сразу все запросы. Чтобы не выполнять запросы один за другим каскадом, мы можем отправлять их *параллельно*, то есть одновременно.

Причина появления каскада запросов заключается в том, что компоненты отображаются внутри друг друга. Компонент `GitHubUser` отображает `UserRepositories`, который в конечном итоге отображает `RepositoryReadme`. Запросы не выполняются, пока каждый их компонент не будет обработан.

Выполнение запросов параллельно требует другого подхода. Сначала удалим `<RepositoryReadme/>` из функции рендеринга `RepoMenu`. Это хороший ход. `RepoMenu` должна заниматься только логистикой создания меню репозиториев, которое пользователь может прокручивать. Компонент `RepositoryReadme` должен быть передан другому компоненту.

Затем удалим `<RepoMenu />` из свойства `renderSuccess` компонента `UserRepositories`. Аналогичным образом `<UserRepositories/>` необходимо удалить из компонента `UserDetails`.

Вместо того чтобы вкладывать компоненты друг в друга, разместим их на одном уровне внутри компонента `App`:

```
import React, { useState } from "react";
import SearchForm from "./SearchForm";
import GitHubUser from "./GitHubUser";
import UserRepositories from "./UserRepositories";
import RepositoryReadme from "./RepositoryReadme";

export default function App() {
  const [login, setLogin] = useState("moonhighway");
  const [repo, setRepo] = useState("learning-react");
  return (
    <>
      <SearchForm value={login} onSearch={setLogin} />
      <GitHubUser login={login} />
      <UserRepositories
        login={login}
        repo={repo}
        onSelect={setRepo}
      />
      <RepositoryReadme login={login} repo={repo} />
    </>
  );
}
```

Компоненты `GitHubUser`, `UserRepositories` и `RepositoryReadme` отправляют HTTP-запросы на GitHub для получения данных. Их выполнение на одном уровне будет происходить параллельно.

Каждому компоненту требуется определенная информация, чтобы сделать запрос. Нам нужен `login`, чтобы получить пользователя GitHub и список пользовательских репозиториев. Для правильной работы `RepositoryReadme` требуется как `login`, так и `repo`. Чтобы убедиться, что у всех компонентов есть все необходимое для выполнения запросов, мы инициализируем приложение для рендеринга сведений о пользователе `moonhighway` и репозитории `learning-react`.

Если пользователь ищет другого `GitHubUser` с помощью `SearchForm`, значение `login` изменится, что вызовет срабатывания хука `useEffect` в компонентах, заставляя их делать дополнительные запросы данных. Если пользователь прокручивает список репозиториев, то будет вызвано свойство `onSelect` компонента `UserRepositories`, что приведет к изменению значения `repo`. Это изменение вызовет хук `useEffect` внутри компонента `RepositoryReadme`, и будет запрошен новый README.

Компонент `RepoMenu` всегда начинается с первого репозитория. Мы должны увидеть, есть ли значение у свойства `selectedRepo`. Если оно есть, используем его, чтобы найти начальный индекс для отображаемого репозитория:

```
export function RepoMenu({ repositories, selected, onSelect = f => f }) {
  const [{ name }, previous, next] = useIterator(
    repositories,
    selected ? repositories.findIndex(repo => repo.name === selected) : null
  );
  ...
}
```

Второй аргумент для хука `useIterator` — это начальный индекс. При наличии свойства `selected` мы будем искать индекс выбранного репозитория по имени. Это поможет убедиться, что в меню репозитория изначально отображается правильный репозиторий. Нам также необходимо передать свойство `selected` компоненту из `UserRepositories`:

```
<Fetch
  uri={`https://api.github.com/users/${login}/repos`}
  renderSuccess={({ data }) => (
    <RepoMenu
      repositories={data}
      selected={repo}
      onSelect={onSelect}
    />
  )}
/>
```

Теперь, когда свойство `repo` передается в `RepoMenu`, нужно выбрать начальный репозиторий, в нашем случае `learning-react`.

Если вы посмотрите на вкладку Network, то заметите, что мы сделали три параллельных запроса (рис. 8.9).

Name	Status	Type	Initiator	Size	Time	Waterfall
moonhighway	200	fetch	hooks.js:11	1.8 KB	2.22 s	
repos	200	fetch	hooks.js:11	14.7 KB	2.57 s	
readme	200	fetch	RepositoryRe...	3.4 KB	2.22 s	
README.md	200	fetch	RepositoryRe...	2.0 KB	2.11 s	

Рис. 8.9. Создание параллельных запросов

Таким образом, компоненты сделали запрос одновременно. Только компонент `RepoReadme` по-прежнему работает каскадно, чтобы получить файл README.

Но ничего страшного. Трудно выполнить каждый запрос правильно при первом рендеринге приложения. Параллельные и каскадные запросы могут работать в команде.

В ожидании значений

Мы задаем значения `login` и `repo` равными `moonhighway` и `learning-react`. Но мы не всегда можем угадать, какие данные отображать в первую очередь. В таком случае мы просто не отображаем компонент, пока не появятся требуемые данные:

```
export default function App() {
  const [login, setLogin] = useState();
  const [repo, setRepo] = useState();
  return (
    <>
      <SearchForm value={login} onSearch={setLogin} />
      {login && <GitHubUser login={login} />}
      {login && (
        <UserRepositories
          login={login}
          repo={repo}
          onSelect={setRepo}
        />
      )}
      {login && repo && (
        <RepositoryReadme login={login} repo={repo} />
      )}
    </>
  );
}
```

В этом сценарии ни один из компонентов не отображается, пока необходимые свойства не получат значения. Первоначально отображается только `SearchForm`. При поиске пользователя значение `login` изменится, что приведет к рендерингу компонента `UserRepositories`. Когда этот компонент ищет репозитории, он выбирает первый репозиторий в списке, вызывая `setRepo`. Наконец, у нас есть `login` и `repo`, поэтому появится компонент `RepositoryReadme`.

Отмена запросов

Пользователь может очистить поле поиска и никого не искать. В этом случае мы хотим убедиться, что значение `repo` тоже будет пустым. Добавим метод `handleSearch`, который гарантирует, что значение `repo` изменится, когда нет значения `login`:

```
export default function App() {
  const [login, setLogin] = useState("moonhighway");
  const [repo, setRepo] = useState("learning-react");

  const handleSearch = login => {
    if (login) return setLogin(login);
    setLogin("");
    setRepo("");
  };

  if (!login)
    return (
      <SearchForm value={login} onSearch={handleSearch} />
    );

  return (
    <>
      <SearchForm value={login} onSearch={handleSearch} />
      <GitHubUser login={login} />
      <UserRepositories
        login={login}
        repo={repo}
        onSelect={setRepo}
      />
      <RepositoryReadme login={login} repo={repo} />
    </>
  );
}
```

Мы добавили метод `handleSearch`. Теперь, когда пользователь очищает поле поиска и ищет пустую строку, значение `repo` также устанавливается на пустую строку. Если по какой-то причине нет значения `login`, мы визуализируем только компонент `SearchForm`. Когда значение `login` есть, мы визуализируем все четыре компонента.

Теперь в нашем приложении технически два экрана. На одном экране отображается только форма поиска. Другой экран отображается только тогда, когда форма поиска содержит значение, и в этом случае отображаются все четыре компонента. Мы договорились включать или отключать компоненты в зависимости от взаимодействия с пользователем. Допустим, нам нужны данные `moon-highway`. Если пользователь очищает поле поиска, компоненты `GitHubUser`, `UserRepositories` и `RepositoryReadme` отключаются и больше не отображаются. Но что, если эти компоненты в момент отключения как раз загружали данные?

Варианты решения:

1. Установить для сети режим Slow 3G, чтобы успеть вызвать проблему.
2. Изменить значение поля поиска с `moonhighway` на `eveporcello`.
3. Пока данные загружаются, выполнять поиск по пустой строке.

Эти действия приведут к отключению `GitHubUser`, `UserRepositories` и `RepositoryReadme`, когда они находятся в процессе выполнения запросов на выборку. Когда будет получен ответ на запрос выборки, эти компоненты отключатся. Попытка изменить значения состояния в отключенном компоненте вызовет ошибку (рис.8.10).

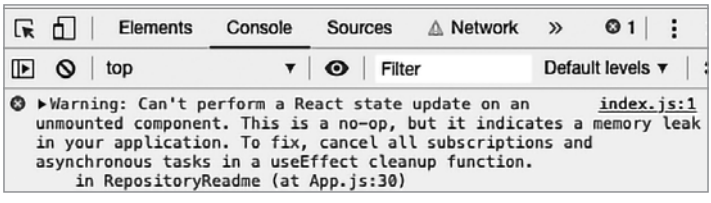


Рис. 8.10. Ошибка при включении компонента

Эти ошибки могут возникать всякий раз, когда пользователь загружает данные через медленную сеть. Но мы можем защитить себя, например создав хук, который сообщит, включен ли текущий компонент:

```
export function useMountedRef() {
  const mounted = useRef(false);
  useEffect(() => {
    mounted.current = true;
    return () => (mounted.current = false);
  });
  return mounted;
}
```

Хук `useMountedRef` использует ссылку. Когда компонент отключается, состояние очищается, но ссылки по-прежнему остаются доступными. Вышеупомянутый хук `useEffect` не имеет массива зависимостей; он вызывается каждый раз при рендеринге компонента и гарантирует, что значение ссылки равно `true`. Каждый раз, когда компонент отключается, вызывается функция, возвращаемая из `useEffect`, которая изменяет значение ссылки на `false`.

Теперь мы можем использовать этот хук внутри компонента `RepoReadme`. Это позволит нам убедиться, что компонент включен, и мы можем применять какие-либо обновления состояния:

```
const mounted = useMountedRef();

const loadReadme = useCallback(async (login, repo) => {
  setLoading(true);
  const uri = `https://api.github.com/repos/${login}/${repo}/readme`;
  const { download_url } = await fetch(uri).then(res =>
    res.json()
```

```
  );
  const markdown = await fetch(download_url).then(res =>
    res.text()
  );
  if (mounted.current) {
    setMarkdown(markdown);
    setLoading(false);
  }
}, []);
```

Теперь у нас есть ссылка, которая сообщает, активен ли компонент. На выполнение обоих запросов потребуется время. Когда все будет готово, мы проверим, что компонент все еще был смонтирован, прежде чем вызывать `setMarkdown` или `setLoading`.

Давайте добавим ту же логику к хуку `useFetch`:

```
const mounted = useMountedRef();

useEffect(() => {
  if (!uri) return;
  if (!mounted.current) return;
  setLoading(true);
  fetch(uri)
    .then(data => {
      if (!mounted.current) throw new Error("component is not mounted");
      return data;
    })
    .then(data => data.json())
    .then(setData)
    .then(() => setLoading(false))
    .catch(error => {
      if (!mounted.current) return;
  setError(error);
});
```

Хук `useFetch` используется для выполнения остальных запросов на выборку в приложении. В этом хуке мы составляем запрос на выборку, используя функции вроде `.then()` вместо `async` и `await`. Когда выборка завершена, мы в первом обратном вызове проверяем, активен ли компонент. Если да, возвращаются данные и вызываются остальные функции `.then`. Если компонент неактивен, первая функция `.then` выдает ошибку, предотвращая выполнение остальных функций `.then`. Затем вызывается функция `.catch`, и в эту функцию передается новая ошибка. Функция `.catch` проверяет, активен ли компонент, до вызова `setError`.

Мы успешно отменили запросы. Мы не останавливали выполнение самого HTTP-запроса, но защитили вызовы состояния, выполняемые после разрешения

запроса. Всегда полезно протестировать приложение в условиях медленной сети. Найденные ошибки будут выявлены и устранены.

Знакомство с GraphQL

Как и React, GraphQL был разработан в Facebook. И как и React, он представляет собой декларативное решение для создания пользовательских интерфейсов и взаимодействия с API. Делая параллельные запросы данных, мы пытаемся получить все необходимые данные одновременно. GraphQL был разработан именно для этого.

Чтобы получить данные из GraphQL API, нам нужно не только сделать HTTP-запрос на определенный URI, но и отправить запрос данных — декларативное описание запрашиваемых данных. Сервис проанализирует это описание и упакует все запрашиваемые данные в один ответ.

GitHub GraphQL API

Чтобы использовать GraphQL в приложении React, серверная служба, с которой вы общаетесь, должна быть построена в соответствии со спецификациями GraphQL. К счастью, у GitHub есть GraphQL API. Большинство сервисов GraphQL предоставляют возможность изучить GraphQL API. На GitHub это GraphQL Explorer (https://docs.github.com/en/graphql/overview/explorer). Чтобы использовать Explorer, войдите в свою учетную запись GitHub.

На левой панели окна Explorer создадим запрос GraphQL. Например, добавим запрос на получение информации об одном пользователе GitHub:

```
query {
  user(login: "moontahoe") {
    id
    login
    name
    location
    avatarUrl
  }
}
```

Это запрос GraphQL. Мы запрашиваем информацию о пользователе GitHub под именем `moontahoe`. Но вместо того, чтобы получать всю общедоступную информацию о нем, мы получаем только те данные, которые нам нужны: `id`, `login`, `avatarUrl`, `name` и `location`. Нажимая кнопку `Play`, мы отправляем этот запрос в виде POST-запроса на https://api.github.com/graphql. Все запросы GitHub

GraphQL отправляются на этот URI. GitHub проанализирует этот запрос и вернет только запрошенные нами данные:

```
{
  "data": {
    "user": {
      "id": "MDQ6VXNlcjU5NTIwODI=",
      "login": "MoonTahoe",
      "name": "Alex Banks",
      "location": "Tahoe City, CA",
      "avatarUrl": "https://github.com/moontahoe.png"
    }
  }
}
```

Мы можем формализовать этот запрос GraphQL в многоразовую операцию под именем `findRepos`. Каждый раз, когда мы хотим найти информацию о пользователе и его репозиториях, мы можем отправить переменную `login` в этот запрос:

```
query findRepos($login: String!) {
  user(login: $login) {
    login
    name
    location
    avatar_url: avatarUrl
    repositories(first: 100) {
      totalCount
      nodes {
        name
      }
    }
  }
}
```

Теперь мы создали формальный запрос `findRepos`, который можно использовать, просто связав значение переменной `$login`. Мы устанавливаем эту переменную с помощью панели `Querry Variables` (рис. 8.11).

Помимо получения сведений о пользователе, мы также запрашиваем первые сто репозиториев этого пользователя. Мы указываем количество репозиториев, возвращенных запросом, `totalCount`, а также имя `name` каждого репозитория. GraphQL возвращает только те данные, которые мы запрашиваем. В данном случае мы получим только имя для каждого репозитория, и больше ничего.

В этот запрос мы внесли еще одно изменение: использовали псевдоним для свойства `avatarUrl`. Поле GraphQL для получения аватара пользователя называется `avatarUrl`, но мы хотим, чтобы эта переменная называлась `avatar_url`. Псевдоним указывает GitHub переименовать это поле в ответе с данными.

GraphQL — огромная тема. Мы написали о нем целую книгу[1] (https://www.oreilly.com/library/view/learning-graphql/9781492030706/). Здесь же мы лишь поверхностно коснулись этой темы и отметим, что GraphQL становится все более востребованным. Чтобы быть успешным разработчиком в XXI веке, важно понимать основы GraphQL.

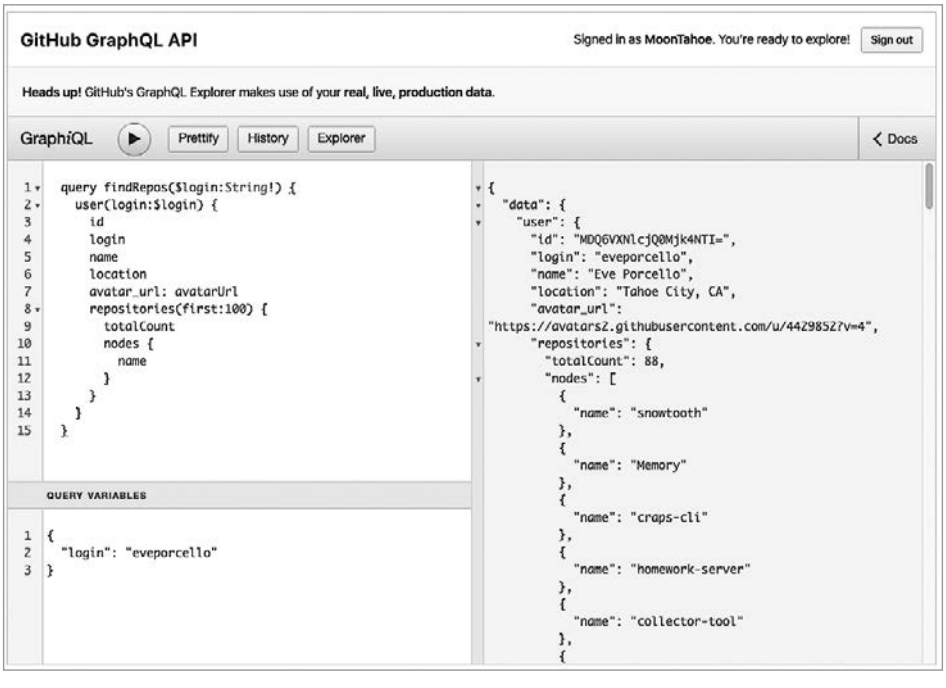


Рис. 8.11. GitHub GraphQL Explorer

Выполнение запроса GraphQL

Запрос GraphQL — это HTTP-запрос, который содержит запрос данных в теле запроса. Вы можете использовать `fetch`, чтобы сделать запрос GraphQL. Также существует ряд библиотек и фреймворков, которые позволяют обрабатывать детали выполнения таких типов запросов. В следующем разделе мы увидим, как заполнить приложение данными GraphQL с помощью библиотеки graphql-request.

[1] *Бэнкс А., Порселло Е.* GraphQL: язык запросов для современных веб-приложений. СПб.: Питер, 2019.

Сфера работы GraphQL не ограничивается протоколом HTTP. Его спецификация выполнения запросов данных по сети подходит для работы с любым сетевым протоколом. Кроме того, GraphQL не зависит от языка.

Для начала установим библиотеку graphql-request:

```
npm i graphql-request
```

GitHub GraphQL API требует идентификации для отправки запросов от клиентских приложений. Чтобы выполнить следующий пример, вы должны получить токен личного доступа от GitHub и отправлять этот токен с каждым запросом.

Чтобы получить личный токен доступа для запросов GraphQL, перейдите в меню Settings > Developer Settings > Personal Access Tokens. В этой форме вы можете создать токен доступа с определенными правами. Для выполнения запросов GraphQL токен должен иметь следующий доступ для чтения:

- user;
- public_repo;
- repo;
- repo_deployment;
- repo:status;
- read:repo_hook;
- read:org;
- read:public_key;
- read:gpg_key.

Можно использовать библиотеку graphql-request для создания запросов GraphQL из JavaScript:

```
import { GraphQLClient } from "graphql-request";

const query = `
  query findRepos($login:String!) {
    user(login:$login) {
      login
      name
      location
      avatar_url: avatarUrl
      repositories(first:100) {
        totalCount
        nodes {
          name
```

```
        }
      }
    }
  }
`;

const client = new GraphQLClient(
  "https://api.github.com/graphql",
  {
    headers: {
      Authorization: `Bearer <PERSONAL_ACCESS_TOKEN>`
    }
  }
);

client
  .request(query, { login: "moontahoe" })
  .then(results => JSON.stringify(results, null, 2))
  .then(console.log)
  .catch(console.error);
```

Этот запрос мы отправляем с помощью конструктора `GraphQLClient` из библиотеки graphql-request. При создании клиента мы используем URI для GitHub GraphQL API: https://api.github.com/graphql. Мы также отправляем дополнительные заголовки, содержащие наш персональный токен доступа. Этот токен идентифицирует нас и требуется GitHub при использовании GraphQL API. Теперь мы можем использовать клиент для выполнения запросов GraphQL.

Чтобы сделать запрос GraphQL, нам понадобится запрос данных `query`. Запрос данных — это просто строка, содержащая запрос GraphQL. Мы отправляем запрос в функцию `request` вместе с любыми переменными, которые могут потребоваться для запроса. В этом случае для `query` требуется переменная с именем `$login`, поэтому мы отправляем объект, содержащий значение `$login` в поле ввода.

Здесь мы просто преобразуем полученный JSON в строку и записываем ее в консоль:

```
{
  "user": {
    "id": "MDQ6VXNlcjU5NTIwODI=",
    "login": "MoonTahoe",
    "name": "Alex Banks",
    "location": "Tahoe City, CA",
    "avatar_url": "https://avatars0.githubusercontent.com/u/5952082?v=4",
    "repositories": {
      "totalCount": 52,
      "nodes": [
        {
```

```
          "name": "snowtooth"
        },
        {
          "name": "Memory"
        },
        {
          "name": "snowtooth-status"
        },

        ...

      ]
    }
  }
}
```

Как и `fetch`, функция `client.request` возвращает промис. Получение этих данных внутри компонента React будет очень похоже на получение данных из маршрута:

```
export default function App() {
  const [login, setLogin] = useState("moontahoe");
  const [userData, setUserData] = useState();
  useEffect(() => {
    client
      .request(query, { login })
      .then(({ user }) => user)
      .then(setUserData)
      .catch(console.error);
  }, [client, query, login]);

  if (!userData) return <p>loading...</p>;

  return (
    <>
      <SearchForm value={login} onSearch={setLogin} />
      <UserDetails {...userData} />
      <p>{userData.repositories.totalCount} — repos</p>
      <List
        data={userData.repositories.nodes}
        renderItem={repo => <span>{repo.name}</span>}
      />
    </>
  );
}
```

Мы делаем запрос `client.request` внутри хука `useEffect`. Если клиент, запрос или логин изменятся, `useEffecthook` сделает другой запрос. Затем мы отрендерим полученный JSON с помощью React (рис. 8.12).

Рис. 8.12. Приложение GraphQL

В этом примере мы не рассматриваем обработку состояний `loading` и `error`, но можем применить все, что мы узнали, к GraphQL. React не волнует, как мы получаем данные. Пока мы понимаем, как работать в компонентах с асинхронными объектами вроде промисов, мы будем готовы ко всему.

Загрузка данных из интернета — это асинхронная задача. Когда мы запрашиваем данные, для их передачи требуется некоторое время, и в процессе что-то может пойти не так. Обработка состояний ожидания, успеха и неудачи промиса в компоненте React представляет собой оркестровку хуков с отслеживанием состояния с помощью хука `useEffect`.

Мы потратили большую часть этой главы на работу с промисами, функцией `fetch` и HTTP. Это связано с тем, что протокол HTTP по-прежнему является самым популярным способом запроса данных из интернета, а промисы хорошо сочетаются с HTTP-запросами. Иногда приходится работать с другими протоко-

лами, например с WebSockets. Не беспокойтесь: вам всегда помогут обработчики состояния и `useEffect`.

Вот краткий пример того, как мы можем включить `socket.io` в настраиваемый хук `useChatRoom`:

```
const reducer = (messages, incomingMessage) => [
  messages,
  ...incomingMessage
];

export function useChatRoom(socket, messages = []) {
  const [status, setStatus] = useState(null);
  const [messages, appendMessage] = useReducer(
    reducer,
    messages
  );

  const send = message => socket.emit("message", message);

  useEffect(() => {
    socket.on("connection", () => setStatus("connected"));
    socket.on("disconnecting", () =>
      setStatus("disconnected")
    );
    socket.on("message", setStatus);
    return () => {
      socket.removeAllListeners("connect");
      socket.removeAllListeners("disconnect");
      socket.removeAllListeners("message");
    };
  }, []);

  return {
    status,
    messages,
    send
  };
}
```

Этот хук предоставляет массив сообщений чата `messages`, статус подключения к веб-сокету `connection` и функцию для трансляции новых сообщений в сокет. На все эти значения влияют слушатели, определенные в хуке `useEffect`. Когда сокет вызывает события `connection` или `disconnecting`, значение `status` изменяется. Новые полученные сообщения добавляются к массиву сообщений с помощью хука `useReducer`.

В этой главе мы обсудили некоторые методы обработки асинхронных данных в приложениях. Это чрезвычайно важная тема, и в следующей главе мы покажем, как использование `Suspense` может привести к будущим изменениям в этой области.

ГЛАВА 9

Suspense

Это наименее важная глава в книге. По крайней мере, так нам сказали члены команды React. Не то чтобы они сказали: «Это ненужная глава, не пишите ее», но опубликовали серию твитов, предупреждающих преподавателей и евангелистов, что большая часть их работы в этой области очень скоро устареет. Все скоро изменится.

Можно сказать, что работа, проделанная командой React с Fiber, Suspense и параллельным режимом, представляет собой будущее веб-разработки. Она может в корне изменить способ интерпретации JavaScript браузерами. Звучит очень серьезно. Мы говорим, что это наименее важная глава этой книги, потому что не хотим, чтобы шумиха вокруг Suspense дала вам ложные ожидания. API-интерфейсы и паттерны, из которых состоит Suspense, не являются единой всеобъемлющей теорией, определяющей работу всего на свете.

Suspense — это просто инструмент. Возможно, вам никогда не понадобится его использовать. Он разработан для решения конкретных проблем, с которыми Facebook сталкивается при масштабной работе. Но наши задачи не такие, как у Facebook, и в этом случае Suspense может излишне усложнить нашу работу. К тому же грядут изменения. Параллельный режим — это экспериментальная функция, и команда React не рекомендует использовать ее в продакшене. Фактически большинство новых концепций подразумевает использование хуков. Если вы не разрабатываете собственные хуки день за днем, вам, вероятно, никогда не понадобится знать о них. Многие техники, связанные с Suspense, можно абстрагировать с помощью хуков.

Хоть мы и убеждали вас в обратном три абзаца, рассматриваемые в этой главе концепции поистине удивительны. При правильном использовании они однажды помогут вам улучшить пользовательский опыт. Если вы ведете или поддерживаете библиотеку хуков и / или компонентов React, эти концепции могут оказаться для вас полезными. Они помогут вам настроить пользовательские хуки так, чтобы обеспечить лучшую обратную связь и расстановку приоритетов.

В этой главе мы создадим еще одно небольшое приложение, чтобы продемонстрировать некоторые из новых функций. По сути мы перестроим приложение из главы 8, воспользовавшись более крупной структурой. Например, используем компонент `SiteLayout`:

```
export default function SiteLayout({
  children,
  menu = c => null
}) {
  return (
    <div className="site-container">
      <div>{menu}</div>
      <div>{children}</div>
    </div>
  );
}
```

Компонент `SiteLayout` будет отображаться в компоненте приложения, чтобы помочь нам составить пользовательский интерфейс:

```
export default function App() {
  return (
    <SiteLayout menu={<p>Menu</p>}>
      <>
        <Callout>Callout</Callout>
        <h1>Contents</h1>
        <p>This is the main part of the example layout</p>
      </>
    </SiteLayout>
  );
}
```

Этот компонент будет использоваться для придания стиля макету (рис. 9.1).

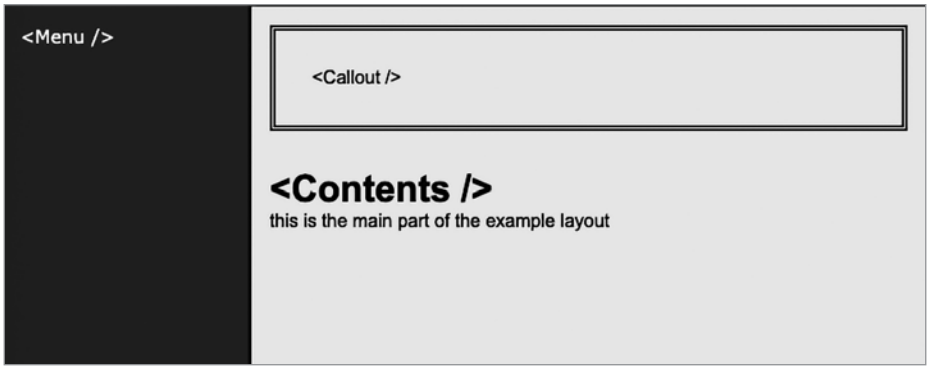


Рис. 9.1. Образец макета

В частности, он позволит нам четко видеть, где и когда отображаются определенные компоненты.

Границы ошибок

Пока что мы не очень внимательно работали с ошибками. Ошибка, возникшая где-либо в дереве компонентов, приведет к остановке всего приложения. Крупные деревья компонентов усложняют проект и затрудняют его отладку. Иногда бывает сложно определить, где произошла ошибка, особенно когда она возникает в компонентах, которые писали не мы.

Границы ошибок — это компоненты, которые можно использовать для предотвращения сбоя всего приложения из-за ошибок. Они также помогают выводить разумные сообщения об ошибках в продакшене. Поскольку ошибки могут обрабатываться одним компонентом, они потенциально могут отслеживать ошибки в приложении и сообщать о них в систему управления проблемами.

В настоящее время единственный способ создать некий ограничивающий ошибки компонент — использовать компонент класса, который тоже скоро будет претерпевать изменения. В будущем создание ограничений для ошибок станет возможным с помощью хука или другого решения, не требующего создания класса. А пока вот пример компонента `ErrorBoundary`:

```
import React, { Component } from "react";

export default class ErrorBoundary extends Component {
  state = { error: null };

  static getDerivedStateFromError(error) {
    return { error };
  }
  render() {
    const { error } = this.state;
    const { children, fallback } = this.props;

    if (error) return <fallback error={error} />;
    return children;
  }
}
```

Это компонент класса. Он хранит состояние, не используя хуки: он получает доступ к определенным методам, которые вызываются в тот или иной момент на протяжении жизненного цикла компонента. Один из таких методов — `getDerivedStateFromError`. Он вызывается, когда в процессе рендеринга где-либо в дочерних элементах происходит ошибка. При возникновении ошибки уста-

навливается значение `state.error`. В случае ошибки отображается компонент `fallback`, и эта ошибка передается компоненту как свойство.

Теперь мы можем использовать этот компонент в дереве, чтобы ловить ошибки и отображать компонент `fallback`, если таковые возникают. Например, мы могли бы обернуть все приложение границей ошибки:

```
function ErrorScreen({ error }) {
  //
  // Here you can handle or track the error before rendering the message
  //

  return (
    <div className="error">
      <h3>We are sorry... something went wrong</h3>
      <p>We cannot process your request at this moment.</p>
      <p>ERROR: {error.message}</p>
    </div>
  );
}

<ErrorBoundary fallback={ErrorScreen}>
  <App />
</ErrorBoundary>;
```

Функция `ErrorScreen` информирует пользователей о том, что произошла ошибка, и отображает информацию о ней. Это дает возможность отслеживать ошибки, возникающие в любом месте приложения. Если в приложении действительно возникает ошибка, вместо черного экрана будет отображаться этот компонент. Мы можем сделать его лучше с помощью небольшого CSS:

```
.error {
  background-color: #efacac;
  border: double 4px darkred;
  color: darkred;
  padding: 1em;
}
```

Чтобы проверить работу компонента, мы собираемся создать компонент, который будет намеренно вызывать ошибки для проверки. Компонент `BreakThings` всегда выдает ошибку:

```
const BreakThings = () => {
  throw new Error("We intentionally broke something");
};
```

Границы ошибок можно комбинировать. Конечно, мы заключили компонент `App` в `ErrorBoundary`, но мы также можем заключить отдельные компоненты в приложение с помощью `Error`:

```
return (
  <SiteLayout
    menu={
      <ErrorBoundary fallback={ErrorScreen}>
        <p>Site Layout Menu</p>
        <BreakThings />
      </ErrorBoundary>
    }
  >
    <ErrorBoundary fallback={ErrorScreen}>
      <Callout>Callout<BreakThings /></Callout>
    </ErrorBoundary>
    <ErrorBoundary fallback={ErrorScreen}>
      <h1>Contents</h1>
      <p>this is the main part of the example layout</p>
    </ErrorBoundary>
  </SiteLayout>
```

Все `ErrorBoundary` будут выполнять резервное копирование, если где-либо в их дочерних элементах возникнет ошибка. В данном случае мы использовали компонент `BreakThings` в меню и внутри `Callout`. Это привело к двойному рендерингу `ErrorScreen` (рис. 9.2).

Мы видим, что `ErrorBoundaries` отображаются правильно. Обратите внимание, что две возникшие ошибки относятся каждая к своей области. Границы подобны стенам, которые не позволяют ошибкам проникать в остальную часть приложения. Несмотря на намеренное создание двух ошибок, контент по-прежнему отображается без проблем.

Рис. 9.2. ErrorBoundaries

На рис. 9.3 видно, что происходит, когда мы перемещаем компонент `BreakThings` только в область контента.

Рис. 9.3. Ошибка

Теперь мы видим, что меню и выноска отображаются без проблем, а в контенте отображается ошибка.

Внутри метода `render` в компоненте класса `ErrorBoundary` мы можем сделать свойство `fallback` необязательным. Когда оно не включено, мы просто будем использовать компонент `ErrorScreen`:

```
render() {
  const { error } = this.state;
  const { children } = this.props;

  if (error && !fallback) return <ErrorScreen error={error} />;
  if (error) return <fallback error={error} />;
  return children;
}
```

Это хорошее решение для последовательной обработки ошибок в приложении. Теперь нам просто нужно обернуть определенные части дерева компонентов с помощью `ErrorBoundary` и позволить компоненту обработать все остальное:

```
<ErrorBoundary>
  <h1>&lt;Contents /&gt;</h1>
  <p>this is the main part of the example layout</p>
  <BreakThings />
</ErrorBoundary>
```

Границы ошибок — это не просто хороший инструмент, но даже необходимое условие удержания пользователей в продакшене, так как они не позволяют небольшой ошибке в относительно несущественном компоненте вывести из строя все приложение.

Разделение кода

Если приложения, над которыми вы сейчас работаете, маленькие, то это, скорее всего, не навсегда. Многие приложения, над которыми вы работаете, в конечном итоге будут содержать массивные кодовые базы с сотнями, может быть, даже тысячами компонентов. Большинство ваших пользователей могут получать доступ к вашим приложениям с телефона в потенциально медленных сетях. Им придется ждать, пока вся кодовая база вашего приложения успешно загрузится, прежде чем React завершит свой первый рендеринг.

Разделение кода дает возможность разделить кодовую базу на управляемые фрагменты, а затем загружать эти фрагменты по мере необходимости. Чтобы продемонстрировать возможности разделения кода, мы добавим в приложение экран пользовательского соглашения:

```
export default function Agreement({ onAgree = f => f }) {
  return (
    <div>
      <p>Terms...</p>
      <p>These are the terms and stuff. Do you agree?</p>
      <button onClick={onAgree}>I agree</button>
    </div>
  );
}
```

Затем переместим остальную часть кодовой базы из компонента App в компонент `Main` и поместим `Main` в отдельный файл:

```
import React from "react";
import ErrorBoundary from "./ErrorBoundary";

const SiteLayout = ({ children, menu = c => null }) => {
  return (
    <div className="site-container">
      <div>{menu}</div>
      <div>{children}</div>
    </div>
  );
};

const Menu = () => (
  <ErrorBoundary>
    <p style={{ color: "white" }}>TODO: Build Menu</p>
  </ErrorBoundary>
);

const Callout = ({ children }) => (
  <ErrorBoundary>
    <div className="callout">{children}</div>
```

```
  </ErrorBoundary>
);

export default function Main() {
  return (
    <SiteLayout menu={<Menu />}>
      <Callout>Welcome to the site</Callout>
      <ErrorBoundary>
        <h1>TODO: Home Page</h1>
        <p>Complete the main contents for this home page</p>
      </ErrorBoundary>
    </SiteLayout>
  );
}
```

Итак, в компоненте `Main` отображается текущий макет сайта. Теперь мы изменим компонент `App`, чтобы он отображал `Agreement` (соглашение), пока пользователь не примет его. Когда он согласится, мы отключим компонент `Agreement` и отобразим компонент `Main`:

```
import React, { useState } from "react";
import Agreement from "./Agreement";
import Main from "./Main";
import "./SiteLayout.css";

export default function App() {
  const [agree, setAgree] = useState(false);

  if (!agree)
    return <Agreement onAgree={() => setAgree(true)} />;

  return <Main />;
}
```

Изначально отображается только компонент `Agreement`. После того как пользователь принимает его условия, значение параметра `agree` изменяется на `true` и отображается компонент `Main`. Проблема в том, что весь код для компонента `Main` и всех его дочерних компонентов упакован в один файл JavaScript: пакет. Это означает, что пользователям придется ждать полной загрузки кодовой базы, и лишь после этого компонент `Agreement` отобразится.

Мы можем отложить загрузку компонента `Main` до его рендеринга, объявив его с помощью `React.lazy` вместо импорта:

```
const Main = React.lazy(() => import("./Main"));
```

Мы говорим React подождать с загрузкой кодовой базы компонента `Main`, пока он не отобразится. После рендеринга он будет импортирован с помощью функции `import`.

Импорт кода во время выполнения аналогичен загрузке данных из интернета. Сначала ожидается запрос кода JavaScript. Он либо выполняется успешно, и тогда файл JavaScript возвращается, либо вызывает ошибку. Нам нужно уведомить пользователя о том, что мы загружаем данные, а также сообщить пользователю, что мы находимся в процессе загрузки кода.

Введение в компонент Suspense

И снова мы оказываемся в ситуации, когда приходится управлять асинхронным запросом. На этот раз нам поможет компонент `Suspense`. Он работает так же, как компонент `ErrorBoundary`. Мы оборачиваем его вокруг определенных компонентов в дереве. Вместо того чтобы возвращаться к сообщению об ошибке при возникновении ошибки, компонент `Suspense` отображает сообщение загрузки, когда происходит отложенная загрузка.

Мы можем изменить приложение для ленивой загрузки компонента `Main` с помощью следующего кода:

```
import React, { useState, Suspense, lazy } from "react";
import Agreement from "./Agreement";
import ClimbingBoxLoader from "react-spinners/ClimbingBoxLoader";

const Main = lazy(() => import("./Main"));

export default function App() {
  const [agree, setAgree] = useState(false);

  if (!agree)
    return <Agreement onAgree={() => setAgree(true)} />;

  return (
    <Suspense fallback={<ClimbingBoxLoader />}>
      <Main />
    </Suspense>
  );
}
```

Теперь приложение в начале работы загружает только кодовую базу React для компонентов `Agreement` и `ClimbingBoxLoader`. React задерживает загрузку компонента `Main` до тех пор, пока пользователь не примет соглашение.

Компонент `Main` заключен в компонент `Suspense`. Как только пользователь принимает соглашение, мы начинаем загружать кодовую базу для компонента `Main`. Поскольку запрос на кодовую базу находится в ожидании, компонент `Suspense` будет отображать `ClimbingBoxLoader` до тех пор, пока база не будет успешно загружена. Как только это произойдет, компонент `Suspense` отключит `ClimbingBoxLoader` и отобразит компонент `Main`.

React Spinners — это библиотека анимированных спиннеров загрузки, которые указывают, что что-то загружается или что приложение работает. В оставшейся части этой главы мы будем пробовать различные компоненты из этой библиотеки. Вы можете установить ее командой `npm i react-spinners`.

А что произойдет, если интернет-соединение оборвется перед попыткой загрузки компонента `Main`? Ошибка. Мы можем избежать ее, заключив компонент `Suspense` в `ErrorBoundary`:

```
<ErrorBoundary fallback={ErrorScreen}>
  <Suspense fallback={<ClimbingBoxLoader />}>
    <Main />
  </Suspense>
</ErrorBoundary>
```

Комбинация этих трех компонентов дает возможность обрабатывать большинство асинхронных запросов. Решение для ожидания: компонент `Suspense` будет отображать анимацию загрузки, пока код не будет загружен. Решение для ошибок: если при загрузке компонента `Main` возникает ошибка, она будет обнаружена и обработана с помощью `ErrorBoundary`. Решение для успеха: если запрос выполнится успешно, мы отобразим компонент `Main`.

Использование Suspense с данными

В предыдущей главе мы создали хук `useFetch` и компонент `Fetch`, который обрабатывает три состояния, связанные с выполнением запроса GitHub: ожидание, успех и неудача. Крутое решение. Однако в последнем разделе мы обработали те же три состояния, используя красивые компоненты `ErrorBoundary` и `Suspense`. Мы делали это для ленивой загрузки кода JavaScript, но мы можем использовать тот же паттерн для задачи загрузки данных.

Допустим, у нас есть компонент `Status`, способный отображать какое-то сообщение о состоянии:

```
import React from "react";

const loadStatus = () => "success — ready";

function Status() {
  const status = loadStatus();
  return <h1>status: {status}</h1>;
}
```

Этот компонент вызывает функцию `loadStatus` для получения сообщения о текущем состоянии. Мы можем отобразить компонент `Status` в компоненте `App`:

```
export default function App() {
  return (
    <ErrorBoundary>
      <Status />
    </ErrorBoundary>
  );
}
```

Если мы запустим этот код как есть, то увидим сообщение об успешном статусе (рис. 9.4).

Рис. 9.4. Успех: все работает

Отображая компонент `Status` в компоненте App, мы следовали хорошей практике разработки React, потому что заключили компонент `Status` в границу ошибки. Теперь, если при загрузке статуса что-то пойдет не так, `ErrorBoundary` вернется к экрану ошибок по умолчанию. Чтобы продемонстрировать это, вызовем ошибку внутри функции `loadStatus`:

```
const loadStatus = () => {
  throw new Error("something went wrong");
};
```

Теперь при запуске мы видим ожидаемый результат. Граница ошибок поймала ошибку и отправила сообщение пользователю (рис. 9.5).

Пока все работает так, как предполагалось. Мы поместили компонент `Status` внутри `ErrorBoundary`, и комбинация этих двух компонентов обрабатывает два из трех состояний промиса: успех или отказ. «Отказ» — это термин, обозначающий сбой или ошибку в промисе.

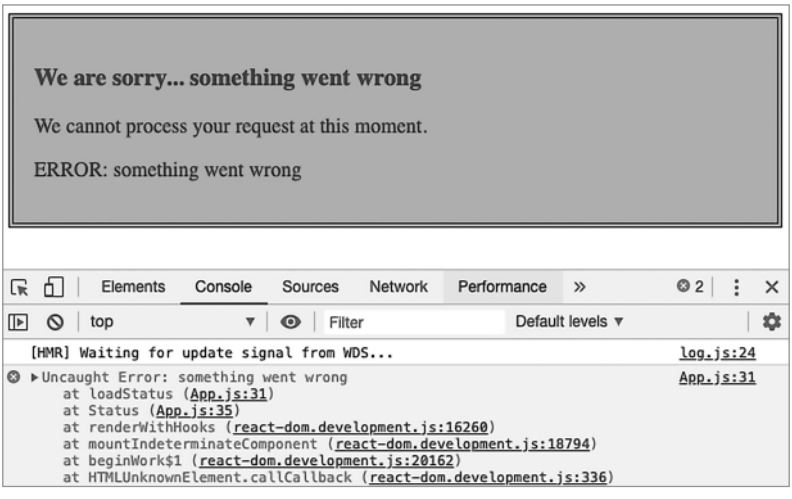


Рис. 9.5. Ошибка поймана в границу

А как насчет ожидания? Это состояние можно запустить, запустив промис:

```
const loadStatus = () => {
  throw new Promise(resolves => null);
};
```

Если мы запустим промис из функции `loadStatus`, то увидим в браузере особую ошибку (рис. 9.6).

Эта ошибка сообщает, что было запущено состояние ожидания, но где-то выше в дереве не настроен компонент `Suspense`. Каждый раз, когда мы запускаем промис из приложения React, нам нужен компонент `Suspense` для обработки отказа:

```
export default function App() {
  return (
    <Suspense fallback={<GridLoader />}>
      <ErrorBoundary>
        <Status />
      </ErrorBoundary>
    </Suspense>
  );
}
```

Теперь у нас есть полноценный набор компонентов для обработки всех трех состояний. Функция `Statusload` по-прежнему выдает промис, но теперь для его обработки есть компонент `Suspense`, настроенный где-то выше в дереве. Запуская промис, мы сообщаем React, что код находится в ожидании. React отвечает, отображая резервный компонент `GridLoader` (рис. 9.7).

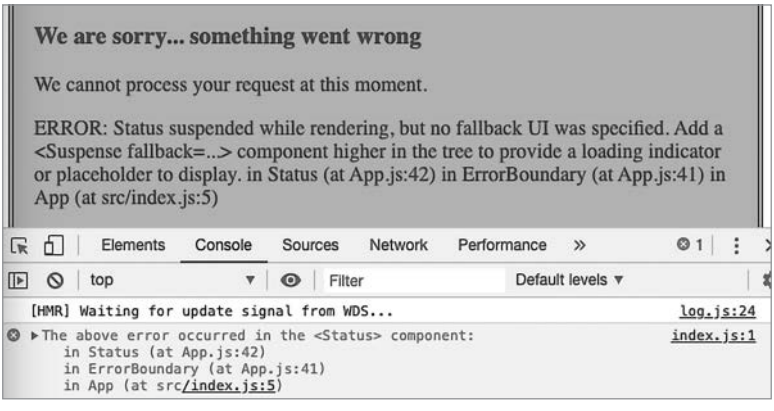


Рис. 9.6. Запуск промиса

Рис. 9.7. GridLoader

Когда функция `loadStatus` успешно возвращает результат, мы отображаем компонент `Status`, как и планировалось. Если что-то пойдет не так (`ifloadStatus` выдаст ошибку), в дело вступит `ErrorBoundary`. Когда `loadStatus` выдает промис, мы запускаем состояние ожидания, которое обрабатывается компонентом `Suspense`.

Это довольно крутой паттерн, но что такое «запустить промис»?

Запуск промиса

В JavaScript ключевое слово `throw` предназначено для ошибок. Вы, наверное, много раз использовали его:

```
throw new Error("inspecting errors");
```

Эта строка кода вызывает ошибку. Если эта ошибка не обрабатывается, происходит сбой всего приложения (рис. 9.8).

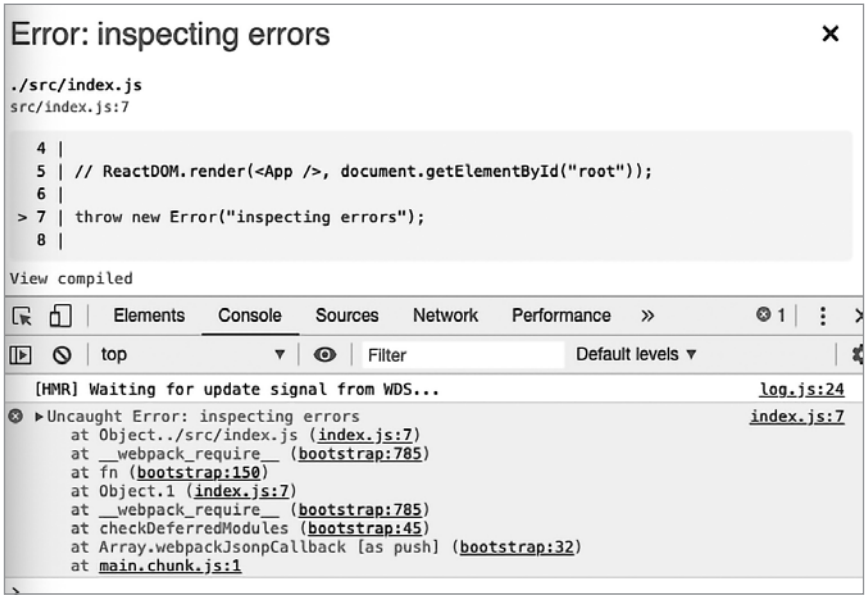


Рис. 9.8. Запуск ошибки

Экран ошибки, отображаемый в браузере, является функцией разработчика Create React App. Когда вы находитесь в режиме разработки, необработанные ошибки выявляются и отображаются прямо на экране. Если вы закроете этот экран, щелкнув «X» в верхнем правом углу, то увидите то, что видят пользователи при возникновении ошибки: пустой белый экран.

Необработанные ошибки всегда видны в консоли. Весь красный текст, который мы видим в консоли, — это информация о запущенной ошибке.

JavaScript — довольно свободный язык. Он позволяет избавиться от множества вещей, от которых мы не можем избавиться при использовании традиционных типизированных языков. Например, в JavaScript мы можем использовать любой тип:

```
throw "inspecting errors";
```

Здесь мы запустили строку. Браузер сообщит, что что-то не было перехвачено, но это не ошибка (рис. 9.9).

Рис. 9.9. GridLoader

На этот раз, когда мы запустили строку, экран с ошибкой Create React App внутри браузера не появился. React знает разницу между ошибкой и строкой.

JavaScript позволяет запускать данные любого типа, в том числе промис:

```
throw new Promise(resolves => null);
```

Теперь браузер сообщает, что что-то не было перехвачено, и это не ошибка, а промис (рис. 9.10).

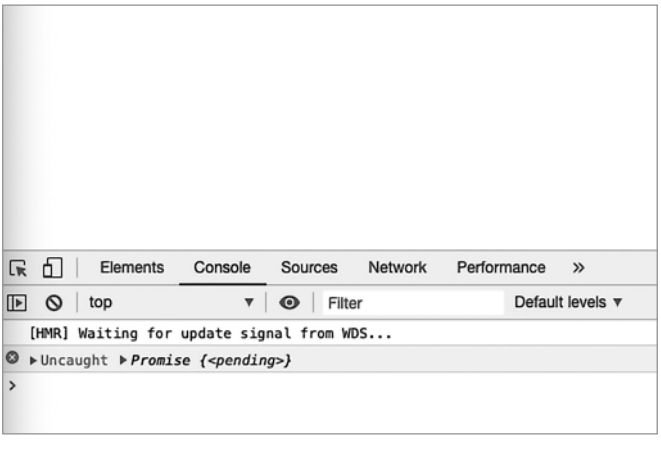


Рис. 9.10. Запуск промиса

Чтобы создать промис в дереве компонентов React, мы сначала делаем это в функции `loadStatus`:

```
const loadStatus = () => {
  console.log("load status");
  throw new Promise(resolves => setTimeout(resolves, 3000));
};
```

Если мы используем функцию `loadStatus` внутри компонента React, генерируется промис, а затем где-то дальше по дереву его захватывает компонент `Suspense`. Так и должно быть: JavaScript позволяет генерировать любой тип, а значит, и перехватить тоже можно любой тип.

Рассмотрим следующий пример:

```
safe(loadStatus);

function safe(fn) {
  try {
    fn();
  } catch (error) {
    if (error instanceof Promise) {
      error.then(() => safe(fn));
    } else {
      throw error;
    }
  }
}
```

Мы отправляем функции `loadStatus` функцию `safe`. И `safe` становится функцией более высокого порядка, а `loadStatus` становится `fn` в рамках функции `safe`. Функция `safe` пытается вызвать функцию `fn`, переданную ей в качестве аргумента. В данном случае `safe` пытается вызвать `loadStatus`. Когда это происходит, `loadStatus` выдает промис с намеренной задержкой в три секунды. Этот промис немедленно перехватывается и становится ошибкой внутри блока `catch`. Мы можем проверить, является ли ошибка промисом, и в данном случае это так. Далее мы ждем выполнения этого промиса, а затем снова пытаемся вызвать `safe` с той же функцией `loadStatus`.

Что мы хотим увидеть, вызывая функцию `safe` рекурсивно с функцией, которая создает промис, вызывающий трехсекундную задержку? Отложенный цикл (рис. 9.11).

Вызывается функция `safe`, промис перехватывается, мы ждем три секунды, пока промис разрешится, затем снова вызываем функцию `safe` с той же функцией, и цикл начинается снова. Каждые три секунды в консоль выводится строка `"load status"`. Сколько раз вы увидите ее — зависит от вашего терпения.

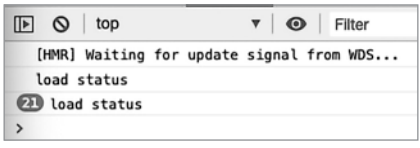


Рис. 9.11. Неудачный цикл

Но мы создавали этот бесконечный рекурсивный цикл не ради проверки вашего терпения, а ради того, чтобы продемонстрировать свою точку зрения. Посмотрите, что происходит, когда мы используем эту новую функцию `loadStatus` в сочетании с предыдущим компонентом `Status`:

```
const loadStatus = () => {
  console.log("load status");
  throw new Promise(resolves => setTimeout(resolves, 3000));
};

function Status() {
  const status = loadStatus();
  return <h1>status: {status}</h1>;
}

export default function App() {
  return (
    <Suspense fallback={<GridLoader />}>
      <ErrorBoundary>
        <Status />
      </ErrorBoundary>
    </Suspense>
  );
}
```

Поскольку `loadStatus` генерирует промис, на экране отображается анимация `GridLoader`. Когда вы смотрите на консоль, результаты снова проверяют ваше терпение (рис. 9.12).

Работает тот же паттерн, что и с функцией `safe`. Компонент `Suspense` знает, что был запущен промис. Отображается компонент `fallback`. Затем компонент `Suspense` ожидает разрешения запущенного промиса, как и функция `safe`. После разрешения промиса компонент `Suspense` повторно отображает компонент `Status`. Когда `Status` снова отображается, он вызывает `loadStatus`, и весь процесс повторяется. На консоль каждые три секунды бесконечно выводится сообщение `"load status"`.

Бесконечный цикл обычно нас не устраивает. И для React тоже не годится. Важно знать, что, когда мы запускаем промис, он перехватывается компонентом `Suspense` и мы переходим в состояние ожидания, пока промис не будет выполнен.

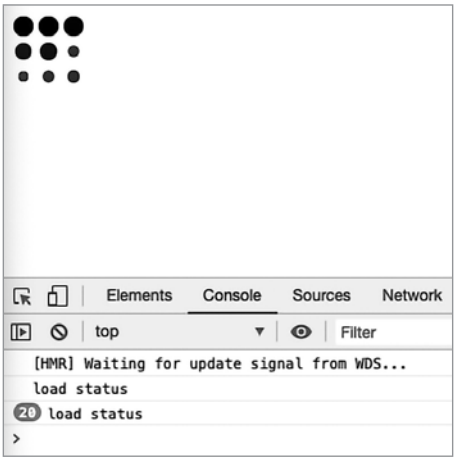


Рис. 9.12. Задержанная рекурсия

Создание источников данных с задержкой

Источник данных с задержкой должен предоставлять данные функции, которая обрабатывает все состояния, связанные с загрузкой: ожидание, успех и ошибка. Функция `loadStatus` может возвращать или генерировать только один тип состояния за раз. Нам нужна функция `loadStatus`, которая будет создавать промис при загрузке данных, возвращать ответ, если данные получены успешно, или выдавать ошибку, если что-то пойдет не так:

```
function loadStatus() {
  if (error) throw error;
  if (response) return response;
  throw promise;
}
```

Найдем место для объявления `error`, `response` и `promise` и убедимся, что эти переменные имеют соответствующую область видимости и не конфликтуют с другими запросами. Для этого определим `loadStatus` с помощью замыкания:

```
const loadStatus = (function() {
  let error, promise, response;

  return function() {
    if (error) throw error;
    if (response) return response;
    throw promise;
  };
})();
```

Это пример замыкания. Область видимости ошибки, промиса и ответа закрыта для любого кода за пределами функции, в которой они определены. Когда мы объявляем функцию `loadStatus`, сразу объявляется и немедленно вызывается анонимная функция: `fn()` совпадает с `(fn)()`. Значение `loadStatus` становится возвращаемой внутренней функцией. Функция `loadStatus` теперь имеет доступ к `error`, `response` и `promise`, но остальная часть нашего JavaScript — нет.

Теперь обработаем значения `error`, `response` и `promise`. `promise` будет отложен в течение трех секунд, прежде чем будет успешно выполнен. После выполнения `promise` значение `response` станет равно «успех». Мы отлавливаем любые ошибки или отказы промисов и используем их для установки значения `error`:

```
const loadStatus = (function() {
  let error, response;
  const promise = new Promise(resolves =>
    setTimeout(resolves, 3000)
  )
    .then(() => (response = "success"))
    .catch(e => (error = e));
  return function() {
    if (error) throw error;
    if (response) return response;
    throw pending;
  };
})();
```

Мы создали промис, ожидающий выполнения три секунды. Если функция `loadStatus` будет вызвана в это время, сам промис будет запущен. По истечении трех секунд промис будет успешно выполнен, и `response` будет присвоено значение. Если вы вызовете `loadStatus` сейчас, она вернет ответ: `success`. Если что-то пойдет не так, функция `loadStatus` вернет `error`.

Функция `loadStatus` — это наш источник данных с задержкой. Он может сообщать свое состояние с помощью архитектуры `Suspense`. Внутренняя работа `loadStatus` жестко запрограммирована. Она всегда разрешает один и тот же промис после трехсекундной задержки. Тем не менее, способ обработки `error`, `response` и `promise` повторяется. Так мы можем обернуть любой промис, чтобы получить источники данных с задержкой.

Для создания источника данных нужен только промис, поэтому мы можем создать функцию, которая принимает промис в качестве аргумента и возвращает источник данных с задержкой. В этом примере мы вызываем функцию `createResource`:

```
const resource = createResource(promise);
const result = resource.read();
```

Предполагается, что `createResource` (промис) успешно создаст объект `resource`. У этого объекта есть функция `read`, и мы можем вызывать ее столько раз, сколько захотим. Когда промис будет разрешен, `read` вернет полученные данные. Когда промис отложен, `read` запустит промис. Если что-то пойдет не так, `read` выдаст ошибку. Этот источник данных готов к работе с задержкой.

Функция `createResource` очень похожа на анонимную функцию, которую мы писали ранее:

```
function createResource(pending) {
  let error, response;
  pending.then(r => (response = r)).catch(e => (error = e));
  return {
    read() {
      if (error) throw error;
      if (response) return response;
      throw pending;
    }
  };
}
```

Эта функция по-прежнему закрывает значения `error` и `response`, но позволяет потребителям передавать промис в качестве аргумента `pending`. Когда ожидающий промис разрешен, мы фиксируем результаты с помощью функции `.then`. Если промис отклонен, мы перехватываем ошибку и используем ее для присвоения значения переменной ошибки.

Функция `createResource` возвращает объект ресурса. Этот объект содержит функцию, называемую `read`. Если промис еще не выполнен, то `error` и `response` не будут определены. Так что `read` запускает промис. Вызов `read`, когда есть значение `error`, вызовет эту ошибку. Наконец, вызов `read` при получении ответа приведет к получению любых данных, разрешенных промисом. Неважно, сколько раз мы вызываем `read` — она всегда точно сообщает состояние промиса.

Чтобы протестировать это в компоненте, нам понадобится промис, в идеале такой, который звучит как название фильма 80-х годов о лыжах:

```
const threeSecondsToGnar = new Promise(resolves =>
  setTimeout(() => resolves({ gnar: "gnarly!" }), 3000)
);
```

Промис `threeSecondsToGnar` ожидает три секунды перед преобразованием в объект, который имеет поле и значение `gnar`. Воспользуемся этим промисом, чтобы создать ресурс данных с задержкой и использовать его в небольшом приложении React:

```
const resource = createResource(threeSecondsToGnar);

function Gnar() {
  const result = resource.read();
  return <h1>Gnar: {result.gnar}</h1>;
}

export default function App() {
  return (
    <Suspense fallback={<GridLoader />}>
      <ErrorBoundary>
        <Gnar />
      </ErrorBoundary>
    </Suspense>
  );
}
```

Компоненты React выполняют много рендеринга. Компонент `Gnar` будет отображен несколько раз, прежде чем фактически вернет ответ. При каждом рендеринге `Gnar` вызывается функция `resource.read()`. При первом рендеринге `Gnar` выдается промис. Этот промис обрабатывается компонентом `Suspense`, после чего отображается компонент `fallback`.

Когда промис будет разрешен, компонент `Suspense` снова попытается отобразить `Gnar`. Тогда `Gnar` снова вызовет `resource.read ()`, но на этот раз, если все прошло нормально, `resource.read ()` успешно вернет `Gnar`, который используется для рендеринга состояния `Gnar` в элементе `h1`. Если что-то пошло не так, `resource.read ()` выдаст ошибку и передаст ее в `ErrorBoundary`.

Как вы понимаете, функцию `createResource` можно сделать довольно надежной. Наш ресурс может попытаться исправить ошибки. Возможно, в случае ошибок в сети ресурс может подождать несколько секунд и автоматически попытаться загрузить данные снова. Он может общаться с другими ресурсами. Возможно, мы сможем регистрировать статистику производительности всех наших ресурсов. Нет предела совершенству. Пока у нас есть функция для чтения текущего состояния ресурса, мы можем делать с самим ресурсом все, что захотим.

Сейчас `Suspense` работает с любым типом асинхронных ресурсов. И мы ожидаем, что это может измениться. Однако каким бы ни был окончательный API для `Suspense`, он обязательно будет обрабатывать три состояния: ожидание, успех и неудача.

Мы рассмотрели `Suspense` API с довольного высокого уровня, и это было сделано намеренно, потому что это экспериментальный материал. Он изменится. Важно почерпнуть из этой главы, что React всегда пытается найти способы сделать приложения быстрее.

За кулисами большой части этой работы стоит то, как работает React, в частности, его алгоритм согласования Fiber.

Fiber

На протяжении всей книги мы говорили о компонентах React как о функциях, возвращающих данные в виде пользовательского интерфейса. Каждый раз, когда эти данные меняются (свойства, состояния, удаленные данные и т. д.), мы возлагаем на React задачу повторного рендеринга компонента. Если мы жмем на звездочку, чтобы поставить оценку, то предполагаем, что пользовательский интерфейс быстро изменится, поскольку этим занимается React. Но как именно это работает? Чтобы понять, как React эффективно обновляет DOM, поговорим подробнее о том, как работает React.

Представьте, что вы пишете статью для блога своей компании. Вам нужна обратная связь, поэтому вы перед публикацией отправляете статью своим коллегам. Они рекомендуют несколько быстрых изменений, и вы создаете совершенно новый документ, печатаете всю статью с нуля, а затем вносите изменения.

Именно с такими дополнительными трудозатратами раньше работали многие библиотеки. Чтобы обновить что-то, приходилось удалять все и начинать с нуля, перестраивая DOM во время обновления.

Второй пример: вы снова отправляете пост для блога своему коллеге. На этот раз ваш коллега создает ветку GitHub, вносит изменения и выполняет слияние, когда все готово. Быстро и эффективно.

Этот процесс похож на работу React. Когда происходит изменение, React создает копию дерева компонентов как объект JavaScript. Затем он ищет части дерева, которые нужно изменить, и изменяет только их. После редактирования копия дерева (рабоча якопия) заменяет существующее дерево. Еще раз подчеркнем: она использует существующие части дерева. Например, вы меняете цвет элемента в списке с красного на зеленый:

```
<ul>
  <li>blue</li>
  <li>purple</li>
  <li>red</li>
</ul>
```

Библиотека React не избавилась от третьего элемента `li`. Вместо этого она заменила дочерние элементы (красный текст) зеленым текстом. Это эффективный подход к обновлению, и именно так React всегда обновляла DOM. Однако здесь есть потенциальная проблема. Обновление DOM — трудозатратная за-

дача, поскольку она выполняется синхронно. Нужно дождаться согласования всех обновлений, а затем их рендеринга, прежде чем выполнять другие задачи в основном потоке. Другими словами, нужно ждать, пока React рекурсивно переместит все обновления, из-за чего пользовательский интерфейс какое-то время не будет работать.

Команда React решила эту проблему, полностью переписав алгоритм согласования React под названием Fiber. Fiber, выпущенный в версии 16.0, задал новую манеру обновления DOM с применением асинхронного подхода. Первым изменением в версии 16.0 было разделение модуля рендеринга и согласователя. Средство визуализации — это часть библиотеки, которая обрабатывает визуализацию, а средство согласования — это часть библиотеки, которая управляет обновлениями, когда они происходят.

Разделение процессов рендеринга и согласования стало большим достижением. Алгоритм согласования хранился в React Core (пакете, устанавливаемом для использования React), и каждая цель рендеринга сама отвечала за рендеринг. Другими словами, сейчас ReactDOM, React Native, React 360 и другие библиотеки отвечают за логику рендеринга и могут быть включены в основной алгоритм согласования React.

Еще одним огромным сдвигом в React Fiber стали изменения в алгоритме согласования. Помните наши дорогие обновления DOM, которые блокировали основной поток? Эта блокировка обновлений и называлась *работой* — но с Fiber React разделяет работу на более мелкие единицы, называемые *волокнами*. Волокно (fiber) — это объект JavaScript, который следит за тем, что он согласовывает и где находится в цикле обновления.

После того как волокно (мельчайшая единица работы) завершено, React связывается с основным потоком, чтобы убедиться, что все задачи выполнены. Если есть важная работа, React передает управление основному потоку. Когда эта важная работа будет завершена, React продолжит обновление. Если в основном потоке нет ничего важного, React переходит к следующему волокну и отображает изменения в DOM.

Вернемся к примеру с GitHub: каждое волокно представляет собой фиксацию в ветке, и когда мы проверяем ветку обратно в основную ветвь, это представляет обновление дерева DOM. Разбивая работу по обновлению на фрагменты, Fiber позволяет приоритетным задачам перемещаться по очереди для немедленной обработки основным потоком. В результате пользовательский интерфейс становится более отзывчивым.

Даже такого функционала Fiber было бы достаточно для успеха, но это еще не все! В дополнение к преимуществам в производительности от разделения

работы на более мелкие блоки перезапись также открывает захватывающие возможности на будущее. В Fiber есть инфраструктура для определения приоритетов обновлений. В более долгосрочной перспективе разработчик может даже настроить параметры по умолчанию и решить, каким типам задач следует дать наивысший приоритет. Процесс приоритезации задач называется *планированием*, и эта концепция лежит в основе экспериментального параллельного режима, который в конечном итоге позволит выполнять задачи параллельно.

Понимание Fiber не является жизненно важным для работы с React в продакшене, но переписывание его алгоритма согласования дает интересное представление о том, как работает и куда стремится React.

ГЛАВА 10

Тестирование в React

Чтобы не отставать от конкурентов, мы должны работать быстро, но не в ущерб качеству. Один из жизненно важных инструментов, который позволяет нам это делать, — *юнит-тестирование*. Оно позволяет проверить каждую часть или модуль приложения на предмет правильной работы[1].

Одним из преимуществ применения функциональных методов программирования является то, что они позволяют писать тестируемый код. Чистые функции сами напрашиваются на тестирование. Неизменяемые вещи легко проверить. Составление приложений из небольших функций, выполняющих отдельные конкретные задачи, позволяет получить тестируемые функции и блоки кода.

В этом разделе мы продемонстрируем методы, которые можно использовать для юнит-тестирования приложений React. В этой главе мы расскажем не только о тестировании, но и об инструментах, которые можно использовать для оценки и улучшения кода и тестов.

ESLint

В большинстве языков программирования код нужно скомпилировать, прежде чем что-либо запускать. У многих языков довольно строгие правила в отношении стиля написания кода для компиляции. У JavaScript таких правил нет, равно как и компилятора. Мы пишем JavaScript-код, скрещиваем пальцы и запускаем его в браузере, чтобы посмотреть, работает он или нет. Хорошо, что у нас есть инструменты для анализа кода, которые могут заставить нас придерживаться определенных правил форматирования.

---

[1] Кратко почитать о юнит-тестировании можно в статье Мартина Фаулера «Unit Testing» (martinfowler.com/bliki/UnitTest.html).

Процесс анализа кода JavaScript называется *хинтингом,* или *линтингом*. JSHint и JSLint — оригинальные инструменты, используемые для анализа JavaScript и получения информации по форматированию. ESLint (eslint.org) — это новейший анализатор кода, поддерживающий синтаксис JavaScript.

Инструмент ESLint является расширяемым. Это означает, что мы можем создавать и публиковать плагины, которые будут добавлены в конфигурацию ESLint для расширения его возможностей.

ESLint прямо из коробки поддерживается приложением Create React, и мы уже видели в консоли его предупреждения о линтах и ошибки.

Мы будем работать с плагином под названием eslint-plugin-react (oreil.ly/3yeXO). Этот плагин будет анализировать наш синтаксис JSX и React вместе с JavaScript.

Установим `eslint` как зависимость разработчика. В этом нам поможет npm:

```
npm install eslint --save-dev

# or

yarn add eslint --dev
```

Прежде чем использовать ESLint, нужно определить правила конфигурации, которым мы будем следовать. Мы определим их в файле конфигурации, который находится в корне проекта. Этот файл можно отформатировать под стиль JSON или YAML. YAML (yaml.org) — это формат данных вроде JSON, у которого меньше синтаксиса, что повышает его удобочитаемость.

ESLint поставляется с инструментом, который помогает настроить конфигурацию. Есть несколько компаний, создававших файлы конфигурации ESLint, которые можно использовать в качестве отправной точки. Но мы можем создать свои файлы.

Чтобы создать конфигурацию ESLint, запустим команду `eslint --init` и ответим на некоторые вопросы о нашем стиле написания кода:

```
npx eslint --init

How would you like to configure ESLint?
To check syntax and find problems

What type of modules does your project use?
JavaScript modules (import/export)

Which framework does your project use?
React

Does your project use TypeScript?
```

```
N

Where does your code run? (Press space to select, a to toggle all, i to invert
selection)
Browser

What format do you want your config file to be in?
JSON

Would you like to install them now with npm?
Y
```

После выполнения команды npx `eslint --initruns` происходят три вещи:

1. eslint-plugin-react устанавливается локально в папку ./node_modules.
2. Зависимости автоматически добавляются в файл package.json.
3. Создается файл конфигурации .eslintrc.json и добавляется в корень проекта.

Если мы откроем этот файл, то увидим перечень настроек:

```
{
  "env": {
    "browser": true,
    "es6": true
  },
  "extends": [
    "eslint:recommended",
    "plugin:react/recommended"
  ],
  "globals": {
    "Atomics": "readonly",
    "SharedArrayBuffer": "readonly"
  },
  "parserOptions": {
    "ecmaFeatures": {
      "jsx": true
    },
    "ecmaVersion": 2018,
    "sourceType": "module"
  },
  "plugins": ["react"],
  "rules": {}
}
```

Если мы посмотрим на ключ `extends`, то увидим, что команда `--init` инициализировала значения по умолчанию для `eslint` и `react`. Это означает, что нам не нужно вручную настраивать все правила. Вместо этого правила предоставляются нам в готовом виде.

Протестируем нашу конфигурацию ESLint и эти правила, создав файл sample.js:

```
const gnar = "gnarly";

const info = ({
  file = __filename,
  dir = __dirname
}) => (
  <p>
    {dir}: {file}
  </p>
);

switch (gnar) {
  default:
    console.log("gnarly");
    break;
}
```

В этом файле есть некоторые проблемы, но нет ничего, что могло бы вызвать ошибки в браузере. Технически этот код работает нормально. Запустим ESLint для этого файла и посмотрим, какие отзывы мы получим на основе наших настроенных правил:

```
npx eslint sample.js

3:7 error 'info' is assigned a value but never used no-unused-vars
4:3 error 'file' is missing in props validation react/prop-types
4:10 error 'filename' is not defined no-undef
5:3 error 'dir' is missing in props validation react/prop-types
5:9 error 'dirname' is not defined no-undef
7:3 error 'React' must be in scope when using JSX react/react-in-jsx-scope

✖ 6 problems (6 errors, 0 warnings)
```

ESLint выполнил статический анализ кода и сообщил о проблемах, определенных согласно выбранной нами конфигурации. Были найдены ошибки при проверке свойств, и еще ESLint не нравится имя файла и имя каталога, потому что он не включает глобальные переменные Node.js автоматически. И, наконец, предупреждения React по умолчанию в ESLint сообщают нам, что React должен быть в области видимости при использовании JSX.

Команда `eslint` выполнит линтинг всего каталога. Для этого нам, вероятно, понадобится, чтобы ESLint игнорировал некоторые файлы JavaScript. Файл .eslintignore позволит перечислить файлы или каталоги, которые ESLint будет пропускать:

```
dist/assets/
sample.js
```

Файл .eslintignore приказывает ESLint игнорировать наш файл sample.js, а также все, что находится в папке dist/assets. Если мы не проигнорируем папку с ресур-

сами, ESLint проанализирует файл клиента bundle.js и, вероятно, найдет в нем много поводов для недовольства.

Добавим в файл package.json сценарий для запуска ESLint:

```
{
  "scripts": {
    "lint": "eslint ."
  }
}
```

Теперь ESLint можно запускать в любое время с помощью команды `npm run lint`, и он проанализирует все файлы в проекте, кроме тех, которые мы проигнорировали.

Плагины ESLint

Есть множество полезных плагинов, которые можно добавить в конфигурацию ESLint. Для проекта React вам обязательно нужно установить eslint-plugin-react-hooks (reactjs.org/docs/hooks-rules.html) — плагин для обеспечения соблюдения правил React Hooks. Этот пакет был выпущен командой React с целью помочь исправить ошибки, связанные с использованием хуков.

Начнем с его установки:

```
npm install eslint-plugin-react-hooks --save-dev

# OR

yarn add eslint-plugin-react-hooks --dev
```

Затем откроем файл .eslintrc.json и добавим следующее:

```
{
  "plugins": [
    // ...
    "react-hooks"
  ],
  "rules": {
    "react-hooks/rules-of-hooks": "error",
    "react-hooks/exhaustive-deps": "warn"
  }
}
```

Этот плагин проверяет, что функции, которые начинаются со слова «use» (предполагается, что это хук), оформлены по правилам хуков.

Добавив это, мы напишем образец кода для тестирования подключаемого модуля. Изменим код в sample.js. Несмотря на то, что этот код не запускается, мы проверяем, правильно ли работает подключаемый модуль:

```
function gnar() {
  const [nickname, setNickname] = useState(
    "dude"
  );
  return <h1>gnarly</h1>;
}
```

Код выдаст несколько ошибок, но, что более важно, среди них есть ошибка, которая сообщает, что мы пытаемся вызвать `useState` в функции, которая не является компонентом или хуком:

```
4:35 error React Hook "useState" is called in function "gnar" that is neither a
React function component nor a custom React Hook function react-hooks/rules-of-
hooks
```

Эти сообщения помогут нам изучить тонкости работы с хуками.

Еще один полезный плагин ESLint — eslint-plugin-jsx-a11y.A11y — это нумероним, означающий, что между буквами «a» и «y» есть 11 букв. Занимаясь вопросами доступности, мы создаем инструменты, веб-сайты и технологии, которые могут использоваться людьми с ограниченными возможностями.

Этот плагин проанализирует код и убедится, что он не нарушает правила доступности. Доступность должна быть в центре внимания, и работа с этим модулем будет способствовать распространению передовых методов при написании доступных приложений React.

Для установки мы снова будем использовать npm или yarn:

```
npm install eslint-plugin-jsx-a11y

// or

yarn add eslint-plugin-jsx-a11y
```

Затем мы добавим в конфигурационный файл .eslintrc.json:

```
{
  "extends": [
    // ...
    "plugin:jsx-a11y/recommended"
  ],
  "plugins": [
    // ...
    "jsx-a11y"
  ]
}
```

Проверим его работу. Мы изменим файл sample.js, включив в него тег изображения, у которого нет свойства `alt`. Чтобы изображение прошло проверку, оно должно иметь свойство `alt` или пустую строку, если изображение не влияет на понимание содержимого пользователем:

```
function Image() {
  return <img src="/img.png" />;
}
```

Если мы снова запустим `lint` с помощью `npm run lint`, то увидим новую ошибку, пойманную плагином jsx / a11y:

```
5:10 error img elements must have an alt prop, either with meaningful text, or
an empty string for decorative images
```

Существует много других подключаемых модулей ESLint для статического анализа кода, и вы можете потратить недели на доведение своей конфигурации ESLint до совершенства. Если вы хотите выйти на новый уровень в разработке, в Awesome ESLint repository (https://github.com/dustinspecker/awesome-eslint) есть много полезных ресурсов.

Prettier

Prettier — это программа для форматирования кода в различных проектах. Сложно переоценить влияние Prettier на повседневную работу веб-разработчиков с момента его выпуска. Если верить историческим записям, споры по поводу синтаксиса занимали 87 % рабочего дня разработчика JavaScript, но теперь есть Prettier, который обрабатывает форматирование кода и определяет правила того, какой синтаксис кода следует использовать в каждом проекте. Это значительно экономит время. Кроме того, если вы когда-либо запускали Prettier на таблице Markdown, выполняемое на глазах быстрое и четкое форматирование наверняка должно было вас впечатлить.

ESLint раньше отвечал за форматирование кода во многих проектах, но теперь есть четкое разграничение обязанностей. ESLint решает проблемы качества кода. Prettier обрабатывает форматирование.

Чтобы Prettier работал с ESLint, мы немного повозимся с конфигурацией проекта. Для начала установим Prettier глобально:

```
sudo npm install -g prettier
```

Теперь мы можем использовать Prettier в любом месте любого проекта.

Настройка Prettier в проекте

Чтобы добавить в проект файл конфигурации Prettier, вы можете создать файл .prettierrc. В этом файле будут описаны настройки проекта по умолчанию:

```
{
  "semi": true,
  "trailingComma": none,
  "singleQuote": false,
  "printWidth": 80
}
```

Это мои значения по умолчанию, но вы, разумеется, можете задать свои. О других параметрах форматирования Prettier можно почитать на странице документации (https://prettier.io/docs/en/options.html).

Заменим то, что сейчас находится в файле sample.js, на код для форматирования:

```
console.log("Prettier Test")
```

Теперь попробуем запустить Prettier CLI из терминала или командной строки:

```
prettier --check "sample.js"
```

Prettier запускает тест и показывает следующее сообщение: `Code style issues found in the above file(s). Forgot to run Prettier?` Чтобы запустить его из интерфейса командной строки, мы можем передать флаг `write`:

```
prettier --write "sample.js"
```

Сделав это, мы увидим некоторое количество миллисекунд, которое потребовалось Prettier для форматирования файла. Если мы откроем файл, то увидим, что содержимое изменилось согласно значениям по умолчанию, указанным в файле .prettierrc. Если этот процесс кажется вам трудоемким, то вы правы. Приступим к автоматизации!

Сначала интегрируем ESLint и Prettier, установив инструмент настройки и плагин:

```
npm install eslint-config-prettier eslint-plugin-prettier --save-dev
```

Конфигурация (eslint-config-prettier) отключает все правила ESLint, которые могут конфликтовать с Prettier. Плагин (eslint-plugin-prettier) интегрирует правила Prettier в правила ESLint. Другими словами, когда мы запустим скрипт `lint`, Prettier тоже будет работать.

Добавим эти инструменты в .eslintrc.json:

```
{
  "extends": [
    // ...
    "plugin:prettier/recommended"
  ],
  "plugins": [
    //,
  "prettier"],
  "rules": {
    // ...
    "prettier/prettier": "error"
  }
}
```

Намеренно нарушим некоторые правила форматирования в своем коде, чтобы Prettier заработал. Например, поправим файл sample.js:

```
console.log("Prettier Test");
```

Выполнение команды `npm run lint` даст следующий результат:

```
1:13 error Replace `'Prettier·Test')` with `"Prettier·Test");` prettier/prettier
```

Все ошибки обнаружены. Теперь вы можете запустить команду записи Prettier и развернуть форматирование одного файла:

```
prettier --write "sample.js"
```

или всех файлов JavaScript в определенных папках:

```
prettier --write "src/*.js"
```

Prettier в VSCode

Если вы используете VSCode, настоятельно рекомендуем настроить Prettier в вашем редакторе. Настройка выполняется довольно быстро и экономит много времени при написании кода.

Сначала установите расширение VSCode для Prettier. Перейдите по ссылке https://oreil.ly/-7Zgz и щелкните Install. После установки запустите Prettier с помощью комбинации клавиш Control + Command + P на Mac или Ctrl + Shift + P на ПК, чтобы вручную отформатировать файл или выделенный фрагмент кода. Также вы сможете форматировать код при сохранении. Для этого потребуется добавить несколько настроек в VSCode.

Чтобы получить доступ к этим настройкам, выберите в меню Code > Preferences > Settings. (Или Command+ на Mac или Ctrl+ на ПК, если вы торопитесь). Затем

щелкните по значку с листом бумаги в правом верхнем углу, чтобы открыть настройки VSCode как JSON. Можно добавить сюда несколько полезных ключей:

```
{
  "editor.formatOnSave": true
}
```

Теперь, когда вы сохраните какой-либо файл, Prettier отформатирует его на основе значений по умолчанию из файла .prettierrc! Просто бомба! Вы также можете выполнить поиск в настройках параметров Prettier, чтобы установить значения по умолчанию в редакторе. Это потребуется для принудительного форматирования, возможного даже при отсутствии файла конфигурации .prettierrc в проекте.

Если вы используете другой редактор, Prettier, вероятно, тоже его поддерживает. Инструкции по работе с другими редакторами кода приведены в разделе Editor Integration в документации (https://prettier.io/docs/en/editors.html).

Проверка типов для приложений React

Когда вы работаете с большим приложением, вам может понадобиться проверка типов для выявления определенных видов ошибок. В приложениях React есть три основных решения для проверки типов: PropTypes, Flow и TypeScript. В следующем разделе мы более подробно рассмотрим, как настроить эти инструменты для повышения качества кода.

PropTypes

На момент первого издания этой книги библиотека PropTypes входила в базовую библиотеку React и была рекомендованным способом добавления проверки типов в приложение. Сегодня, в связи с появлением других решений, таких как Flow и TypeScript, ее функциональность была перенесена в отдельную библиотеку, чтобы уменьшить размер пакета React. Тем не менее PropTypes — широко используемое решение.

Чтобы добавить PropTypes в приложение, установите библиотеку prop-types:

```
npm install prop-types --save-dev
```

Проверим работу, создав минимальный компонент App, который отображает имя библиотеки:

```
import React from "react";
import ReactDOM from "react-dom";
```

```
function App({ name }) {
  return (
    <div>
      <h1>{name}</h1>
    </div>
  );
}

ReactDOM.render(
  <App name="React" />,
  document.getElementById("root")
);
```

Затем мы импортируем библиотеку типов свойств и используем `App.propTypes` для определения типа каждого свойства:

```
import PropTypes from "prop-types";

function App({ name }) {
  return (
    <div>
      <h1>{name}</h1>
    </div>
  );
}

App.propTypes = {
  name: PropTypes.string
};
```

Компонент `App` имеет одно имя свойства и всегда должен быть строкой. Если в качестве имени передано неверное значение типа, будет выдана ошибка. Например, если мы использовали логическое значение:

```
ReactDOM.render(
  <App name="React" />,
  document.getElementById("root")
);
```

консоль сообщит о проблеме:

```
Warning: Failed prop type: Invalid prop name of type boolean supplied to App,
expected string. in App
```

Если для свойства указано значение неправильного типа, предупреждение появляется только в режиме разработки. Предупреждения и некорректные рендеры не появятся в продакшене.

Разумеется, проверять можно и другие типы. Мы могли бы добавить логическое значение, указывающее, используется ли технология в компании:

```
function App({ name, using }) {
  return (
    <div>
      <h1>{name}</h1>
      <p>
        {using ? "used here" : "not used here"}
      </p>
    </div>
  );
}

App.propTypes = {
  name: PropTypes.string,
  using: PropTypes.bool
};

ReactDOM.render(
  <App name="React" using={true} />,
  document.getElementById("root")
);
```

Вот список проверяемых типов:

- `PropTypes.array`
- `PropTypes.object`
- `PropTypes.bool`
- `PropTypes.func`
- `PropTypes.number`
- `PropTypes.string`
- `PropTypes.symbol`

Кроме того, если вы хотите узнать, есть ли у объекта значение, то можете добавить `.isRequired` в конце любого из этих параметров. Например, чтобы задать строку, используйте такой код проверки:

```
App.propTypes = {
  name: PropTypes.string.isRequired
};

ReactDOM.render(
  <App />,
  document.getElementById("root")
);
```

Если вы не укажете значение для этого поля, в консоли появится следующее предупреждение:

```
index.js:1 Warning: Failed prop type: The prop name is marked as required in
App, but its value is undefined.
```

Также возможны ситуации, когда вам все равно, каково значение, если оно указано. В этом случае вы можете использовать любой тип. Например:

```
App.propTypes = {
  name: PropTypes.any.isRequired
};
```

Это означает, что можно указать логическое значение, строку, число — что угодно. Поскольку переменная `name` не определена, проверка типов завершится успешно.

В дополнение к базовой проверке типов существует еще несколько утилит, которые полезны во многих реальных ситуациях. Рассмотрим компонент, в котором есть два варианта статуса: `Open` и `Closed`:

```
function App({ status }) {
  return (
    <div>
      <h1>
        We're {status === "Open" ? "Open!" : "Closed!"}
      </h1>
    </div>
  );
}

ReactDOM.render(
  <App status="Open" />,
  document.getElementById("root")
);
```

Состояние — это строка, поэтому мы можем использовать проверку строки:

```
App.propTypes = {
  status: PropTypes.string.isRequired
};
```

Этот код работает хорошо, но если будут переданы другие строковые значения, отличные от `Open` и `Closed`, свойство будет проверено. Нам нужна проверка перечисляемого типа. Перечисляемый тип — это ограниченный список параметров для определенного поля или свойства. Настроим объект `propTypes`:

```
App.propTypes = {
  status: PropTypes.oneOf(["Open", "Closed"])
};
```

Теперь, если в переменную попадет что-либо, отличное от значений из массива, переданного в `PropTypes.oneOf`, появится предупреждение.

Чтобы узнать обо всех параметрах, которые вы можете настроить для PropTypes в приложении React, ознакомьтесь с документацией (oreil.ly/pO2Js).

Flow

Flow — это библиотека проверки типов, которая используется и поддерживается командой Facebook Open Source. Это инструмент, который проверяет наличие ошибок с помощью аннотаций статического типа. Другими словами, если вы создаете переменную определенного типа, Flow проверит, правильного ли типа используемое значение.

Запустим проект Create React App:

```
npx create-react-app in-the-flow
```

Затем мы добавим в проект Flow. Create React App изначально не предполагает, что вы хотите использовать Flow, поэтому встроенной библиотеки нет, но это легко исправить:

```
npm install --save flow-bin
```

После установки мы добавим сценарий npm для запуска Flow при вводе `npm run flow`. В файл package.json добавим вот такой код:

```
{
  "scripts": {
    "start": "react-scripts start",
    "build": "react-scripts build",
    "test": "react-scripts test",
    "eject": "react-scripts eject",
    "flow": "flow"
  }
}
```

Теперь запуск команды `flow` запустит проверку типов наших файлов. Однако, прежде чем это использовать, создадим файл .flowconfig. Для этого запустим команду:

```
npm run flow init
```

Это создаст скелет файла конфигурации, который будет выглядеть следующим образом:

```
[ignore]

[include]

[libs]
```

```
[lints]

[options]

[strict]
```

В большинстве случаев это поле остается пустым, чтобы использовать настройки Flow по умолчанию. Если вы хотите настроить Flow на что-то более сложное, посмотрите дополнительные параметры в документации (flow.org/en/docs/config/).

Одна из самых крутых особенностей Flow — это возможность постепенного внедрения. Добавление проверки типов ко всему проекту может показаться утомительным. Но для Flow это не обязательно. Достаточно добавить строку `//@flow` в начало файлов, которые вы хотите проверить, и Flow будет автоматически проверять только эти файлы.

Другой вариант — добавить расширение VSCode для Flow, которое помогает завершить код и выдает подсказки по параметрам. Если у вас настроен Prettier или инструмент линтинга, это поможет редактору справиться с неожиданным синтаксисом Flow. Вы можете найти VSCode в магазине https://oreil.ly/zdaPv.

Откроем файл index.js и для простоты сохраним все в одном файле. Не забудьте добавить `//@flow` в начало файла:

```
//@flow

import React from "react";
import ReactDOM from "react-dom";

function App(props) {
  return (
    <div>
      <h1>{props.item}</h1>
    </div>
  );
}

ReactDOM.render(
  <App item="jacket" />,
  document.getElementById("root")
);
```

Определим типы для свойств:

```
type Props = {
  item: string
};

function App(props: Props) {
  //...
}
```

Затем запустим Flow командой `npm run flow`. В некоторых версиях Flow есть это предупреждение:

```
Cannot call ReactDOM.render with root bound to container because null [1] is
incompatible with Element [2]
```

Оно сообщает, что, если функция `document.getElementById("root")` вернет значение `null`, приложение выйдет из строя. Чтобы защититься от этого (и устранить ошибку), можно сделать одно из двух. Первое — использовать оператор `if`, чтобы убедиться, что `root` не равен `null`:

```
const root = document.getElementById("root");

if (root !== null) {
  ReactDOM.render(<App item="jacket" />, root);
}
```

Второе — добавить проверку типа к корневой константе с использованием синтаксиса Flow:

```
const root = document.getElementById("root");

ReactDOM.render(<App item="jacket" />, root);
```

Оба способа позволяют убрать ошибку и убедиться, что код не содержит ошибок!

```
Без ошибок!
```

Мы могли бы довериться такому решению, но вместо этого попробуем его сломать. Передадим в приложение другой тип свойства:

```
ReactDOM.render(<App item={3} />, root);
```

Круто, мы все сломали! И получили сообщение об ошибке:

```
Cannot create App element because number [1] is incompatible with string [2] in
property item.
```

Теперь вернем все обратно, добавим еще одно свойство для числа и скорректируем определения компонентов и свойств:

```
type Props = {
  item: string,
  cost: number
};

function App(props: Props) {
  return (
    <div>
      <h1>{props.item}</h1>
      <p>Cost: {props.cost}</p>
```

```
    </div>
  );
}

ReactDOM.render(
  <App item="jacket" cost={249} />,
  root
);
```

Пока работает. А что, если удалить стоимость?

```
ReactDOM.render(<App item="jacket" />, root);
```

Сразу получим ошибку:

```
Cannot create App element because property cost is missing in props [1] but
exists in Props [2].
```

Если стоимость правда не является обязательным значением, мы можем сделать его необязательным в определениях свойств, используя вопросительный знак после имени свойства — `cost?`:

```
type Props = {
  item: string,
  cost?: number
};
```

Если мы запустим приложение снова, ошибки не будет.

Это верхушка айсберга со всеми функциями, которые предлагает Flow. Чтобы узнать больше и быть в курсе изменений в библиотеке, загляните на официальный сайт flow.org/en/docs/getting-started/.

TypeScript

TypeScript — еще один популярный инструмент для проверки типов в приложениях React. Он относится к надмножеству JavaScript с открытым исходным кодом, то есть добавляет языку дополнительные функции. TypeScript, созданный в Microsoft, предназначен для использования в больших приложениях, чтобы помочь разработчикам находить ошибки и быстрее выполнять новый этап в проекте.

Число сторонников TypeScript растет, поэтому инструменты в его экосистеме продолжают улучшаться. Один из инструментов, с которым мы уже знакомы, — это приложение Create React. В нем есть шаблон TypeScript, который мы можем использовать. Настроим базовую проверку типов, аналогичную тем, которые мы сделали с PropTypes и Flow, чтобы понять, как ее использовать в приложениях.

Начнем с создания еще одного приложения Create React, на этот раз с другими флагами:

```
npx create-react-app my-type --template typescript
```

Теперь познакомимся с особенностями заготовки проекта. Обратите внимание, что в каталоге src файлы теперь имеют расширения .ts или .tsx. Там же есть файл .tsconfig.json, который содержит все наши настройки TypeScript. Подробнее об этом чуть позже.

Кроме того, если вы посмотрите на файл package.json, то увидите там новые зависимости, связанные с TypeScript — сама библиотека и определения типов для Jest, React, ReactDOM и других библиотек. Любая зависимость, которая начинается с `@types/`, описывает определения типов для библиотеки. Это означает, что функции и методы в библиотеке типизированы и нам не нужно описывать все типы библиотеки.

Если ваш проект не включает функции TypeScript, возможно, вы используете старую версию приложения Create React. Чтобы избавиться от этого, запустите команду `npm uninstall -g create-response-app`.

Попробуем перетащить компонент из раздела «Flow» в проект. Добавим в файл index.ts следующее:

```
import React from "react";
import ReactDOM from "react-dom";

function App(props) {
  return (
    <div>
      <h1>{props.item}</h1>
    </div>
  );
}

ReactDOM.render(
  <App item="jacket" />,
  document.getElementById("root")
);
```

Если мы запустим проект командой `npm start`, то увидим первую ошибку TypeScript. Ожидаемо:

```
Parameter 'props' implicitly has an 'any' type.
```

Это означает, что нужно добавить правила типов для компонента App. Начнем с определения типов, которое делается так же, как во Flow. item является строкой, поэтому мы добавим ее к типу AppProps:

```
type AppProps = {
  item: string;
};

ReactDOM.render(
  <App item="jacket" />,
  document.getElementById("root")
);
```

Затем мы будем ссылаться на AppProps в компоненте:

```
function App(props: AppProps) {
  return (
    <div>
      <h1>{props.item}</h1>
    </div>
  );
}
```

Теперь компонент будет отображаться без проблем с TypeScript. Мы также можем деструктурировать реквизиты, если хотим:

```
function App({ item }: AppProps) {
  return (
    <div>
      <h1>{item}</h1>
    </div>
  );
}
```

Этот код можно сломать передачей значения другого типа в качестве свойства элемента:

```
ReactDOM.render(
  <App item={1} />,
  document.getElementById("root")
);
```

Это сразу вызовет ошибку:

```
Type 'number' is not assignable to type 'string'.
```

Ошибка сообщит, в какой строке возникла проблема. Это чрезвычайно полезно при отладке.

Однако TypeScript помогает не только при проверке свойств. Мы можем использовать вывод типа TypeScript, чтобы выполнить проверку типов для значений хука.

Рассмотрим значение состояния для `fabricColor` с начальным состоянием `purple`. Компонент может выглядеть так:

```
type AppProps = {
  item: string;
};

function App({ item }: AppProps) {
  const [fabricColor, setFabricColor] = useState(
    "purple"
  );
  return (
    <div>
      <h1>
        {fabricColor} {item}
      </h1>
      <button
        onClick={() => setFabricColor("blue")}
      >
        Make the Jacket Blue
      </button>
    </div>
  );
}
```

Обратите внимание, что мы не добавили ничего к объекту определений типов. Вместо этого TypeScript предполагает, что тип `fabricColor` соответствует типу его начального состояния. Если мы попробуем установить для свойства `fabricColor` число вместо строки, появится ошибка:

```
<button onClick={() => setFabricColor(3)}>
```

Ошибка выглядит так:

```
Argument of type '3' is not assignable to parameter of type string.
```

TypeScript позволяет выполнять довольно простую проверку типов значения. Конечно, вы можете настроить все сами дополнительно, но то, что мы рассмотрели, может стать хорошим началом.

Чтобы узнать больше о TypeScript, ознакомьтесь с официальной документацией (oreil.ly/97_Px) и шпаргалками по скриптам на GitHub (oreil.ly/vmran).

Разработка через тестирование

Разработка через тестирование или TDD — это практика, а не технология. И она не означает, что у приложения просто должны быть тесты. Скорее это практика, в которой тесты управляют процессом разработки. Чтобы практиковать TDD, нужно выполнять следующее:

Сначала писать тесты

Это самый ответственный шаг. Сначала объявляйте, что хотите сделать и как это должно работать. Этапы тестирования будут называться: красный, зеленый и золотой.

Запускать тесты и проверять, что они не работают (красный)

Прежде чем писать код, запустите тесты и убедитесь, что ничего не работает.

Писать минимальный объем кода, необходимый для прохождения теста (зеленый)

Сосредоточьтесь на выполнении каждого теста и не добавляйте функции, выходящие за рамки теста.

Проводить рефакторинг кода и тестов (золотой)

После того как тесты будут пройдены, внимательнее рассмотрите код и тесты. Постарайтесь изложить код как можно проще и элегантнее.[1]

Концепция TDD дает нам отличный подход к разработке приложения React, особенно при тестировании хуков. Как правило, проще понять, как должен работать хук, чем написать его. Практика TDD позволяет построить и сертифицировать всю структуру данных для функции или приложения независимо от пользовательского интерфейса.

TDD и обучение

Если вы новичок в TDD или плохо знакомы с языком, который тестируете, написать тест до написания кода может показаться вам трудной задачей. Это нормально, и поначалу можно писать код перед тестом, пока вы не освоитесь. Попробуйте работать небольшими партиями: немного кода, несколько тестов и так далее. Когда вы привыкнете к написанию тестов, проще будет начинать с них.

---

[1] Подробнее об этом паттерне разработки см. работы Джеффа МакВертера и Джеймса Бендера. «Red, Green, Refactor» (oreil.ly/Hr6Me)

В оставшейся части этой главы мы будем писать тесты для уже существующего кода, то есть не будем практиковать TDD. Однако в следующем разделе мы представим, что код еще не написан, чтобы прочувствовать рабочий процесс TDD.

Внедрение Jest

Прежде чем писать тесты, нужно выбрать среду тестирования. Вы можете писать тесты для React с любой средой тестирования JavaScript, но официальная документация React рекомендует выполнять тестирование с помощью Jest — средства запуска тестов JavaScript, которое позволяет получить доступ к DOM через JSDOM. Доступ к DOM важен, потому что нам нужна возможность проверять, что выводит React и правильно ли работает приложение.

Create React App и тестирование

Проекты, созданные с помощью Create React App, включают пакет `jest`. Мы можем создать еще один проект Create React App, чтобы начать работу, или использовать существующий:

```
npx create-react-app testing
```

Начнем разговор о тестировании с небольшого примера. Создадим в папке src два новых файла: functions.js и functions.test.js. Помните, что Jest уже сконфигурирован и установлен в Create React App, поэтому мы можем сразу начать писать тесты. В файле functions.test.js зададим тесты. Другими словами, мы напишем, что, по нашему мнению, должна делать функция.

Мы хотим, чтобы функция принимала значение, умножала его на два и возвращала результат. Смоделируем это в тесте. Тестовая функция — это функция, которую Jest использует для проверки одного действия:

```
test("Multiplies by two", () => {
  expect();
});
```

Первый аргумент `Multiplies by two` — это имя теста. Второй аргумент — это функция, которая содержит то, что должно быть проверено, а третий (необязательный) аргумент задает тайм-аут. По умолчанию он составляет пять секунд.

Далее зададим функцию, которая будет умножать числа на два. Она станет нашей тестируемой системой (SUT). В файле functions.js создадим функцию:

```
export default function timesTwo() {...}
```

Экспортируем ее, чтобы использовать SUT в тесте. В тестовом файле мы хотим импортировать функцию и будем использовать оператор `expect` для записи утверждения. В утверждении мы укажем, что, если мы передадим функции `timesTwo` число 4, то ожидаем, что она вернет `8`:

```
import { timesTwo } from "./functions";

test("Multiplies by two", () => {
  expect(timesTwo(4)).toBe(8);
});
```

Сопоставители Jest возвращаются функцией `expect` и используются для проверки результатов. Чтобы протестировать функцию, мы будем использовать сопоставитель `.toBe`. Он подтвердит, что полученный объект соответствует аргументу, отправленному в `.toBe`.

Запустим тесты и посмотрим, как они не работают, используя команду `npm test` или `npm run test`. Jest предоставит конкретную информацию о каждом сбое, включая трассировку стека:

```
FAIL src/functions.test.js
  × Multiplies by two (5ms)

  • Multiplies by two

    expect(received).toBe(expected) // Object.is equality

    Expected: 8
    Received: undefined

      2 |
      3 | test("Multiplies by two", () => {
    > 4 |   expect(timesTwo(4)).toBe(8);
        |                       ^
      5 | });
      6 |

      at Object.<anonymous> (src/functions.test.js:4:23)

Test Suites: 1 failed, 1 total
Tests:       1 failed, 1 total
Snapshots:   0 total
Time:        1.048s
Ran all test suites related to changed files.
```

Потратив время на написание тестов и их запуск, мы увидели, что они не пройдены, то есть работают, как задумано. Отчет о сбое представляет собой список задач. Наша работа — написать минимальный код, необходимый для прохождения тестов.

Теперь, если мы добавим в файл functions.js необходимую функциональность, то сможем пройти тесты:

```
export function timesTwo(a) {
  return a * 2;
}
```

Сопоставитель `.toBe` помог проверить равенство с одним значением. Если мы хотим протестировать объект или массив, то можем использовать сопоставитель `.toEqual`. Выполним еще одну задачу с тестами. В тестовом файле проверим равенство массивов объектов.

У нас есть список блюд из ресторана Guy Fieri в Лас-Вегасе. Нужно создать объект из заказанных товаров, чтобы покупатель получил то, что он хочет, и знал, сколько он должен заплатить. Сначала напишем тест:

```
test("Build an order object", () => {
  expect();
});
```

Затем функцию:

```
export function order(items) {
  // ...
}
```

Теперь воспользуемся функцией заказа в тестовом файле. Предположим, что у нас есть начальный список данных для заказа, который нужно преобразовать:

```
import { timesTwo, order } from "./functions";

const menuItems = [
  {
    id: "1",
    name: "Tatted Up Turkey Burger",
    price: 19.5
  },
  {
    id: "2",
    name: "Lobster Lollipops",
    price: 16.5
  },
  {
    id: "3",
    name: "Motley Que Pulled Pork Sandwich",
    price: 21.5
  },
  {
    id: "4",
    name: "Trash Can Nachos",
```

```
    price: 19.5
  }
];

test("Build an order object", () => {
  expect(order(menuItems));
});
```

Помните, что мы будем использовать `toEqual`, потому что проверяем значение объекта, а не массива. Чему должен быть равен результат? Мы хотим создать объект, который будет выглядеть так:

```
const result = {
  orderItems: menuItems,
  total: 77
};
```

Поэтому мы просто добавим это в тест и используем в утверждении:

```
test("Build an order object", () => {
  const result = {
    orderItems: menuItems,
    total: 77
  };
  expect(order(menuItems)).toEqual(result);
});
```

Теперь допишем функцию в файле functions.js:

```
export function order(items) {
  const total = items.reduce(
    (price, item) => price + item.price,
    0
  );
  return {
    orderItems: items,
    total
  };
}
```

Открыв консоль, мы обнаружим, что тесты пройдены! Пример может показаться тривиальным, но если вы извлекаете данные, вполне вероятно, что вы будете проверять совпадение форм массивов и объектов.

Другая часто используемая с Jest функция — `description()`. Если вы использовали другие библиотеки тестирования, возможно, вы уже видели подобную функцию раньше. Эта функция обычно используется для обертывания нескольких связанных тестов. Например, если бы у нас было несколько тестов для аналогичных функций, мы могли бы заключить их в оператор `describe`:

```
describe("Math functions", () => {
  test("Multiplies by two", () => {
    expect(timesTwo(4)).toBe(8);
  });
  test("Adds two numbers", () => {
    expect(sum(4, 2)).toBe(6);
  });
  test("Subtracts two numbers", () => {
    expect(subtract(4, 2)).toBe(2);
  });
});
```

Когда вы заключаете тесты в оператор `describe`, средство выполнения тестов создает блок тестов, благодаря чему результаты тестирования в терминале выглядят более организованными и удобными для чтения:

```
Math functions
    ✓ Multiplies by two
    ✓ Adds two numbers
    ✓ Subtracts two numbers (1ms)
```

По мере роста числа тестов их группировка в блоки может быть полезным улучшением.

Этот типичный цикл TDD. Сначала мы написали тесты, а затем код, который их проходит. После прохождения тестов мы можем более внимательно изучить код, чтобы увидеть, есть ли в нем места для рефакторинга. Такой подход очень эффективен при работе с JavaScript (и вообще с любым языком).

Тестирование компонентов React

Теперь, когда у нас есть базовое представление о процессе написания тестов, мы можем начать тестировать компоненты React.

Компоненты React — это инструкции, которым React следует при создании обновлений DOM и управлении ими. Мы можем протестировать компоненты, отобразив их и проверив получившуюся DOM.

Мы запускаем тесты не в браузере, а в терминале с Node.js. У Node.js нет DOM API, который входит в стандартную комплектацию каждого браузера. Jest включает в себя пакет npm под названием `jsdom`, который используется для моделирования среды браузера в Node.js, что необходимо для тестирования компонентов React.

Для каждого теста компонента нам, вероятно, потребуется отобразить дерево компонентов React в элемент DOM. Чтобы продемонстрировать этот рабочий процесс, вернемся к компоненту `Star` из файла Star.js:

```
import { FaStar } from "react-icons/fa";

export default function Star({ selected = false }) {
  return (
    <FaStar color={selected ? "red" : "grey"} id="star" />
  );
}
```

Затем в файле index.js импортируем и отобразим звезду:

```
import Star from "./Star";

ReactDOM.render(
  <Star />,
  document.getElementById("root")
);
```

Теперь напишем тест. Мы уже написали код для звезды, поэтому процесса TDD тут не будет. Если бы вам пришлось включить тесты в существующие приложения, вы бы так и делали. В новом файле Star.test.js начнем с импорта React, ReactDOM и `Star`:

```
import React from "react";
import ReactDOM from "react-dom";
import Star from "./Star";

test("renders a star", () => {
  const div = document.createElement("div");
  ReactDOM.render(<Star />, div);
});
```

Перейдем к написанию тестов. Помните, что первый аргумент, который мы предоставляем для проверки, — это имя теста. Затем мы собираемся поработать напильником и создать блок `div`, в который мы можем отобразить звезду с помощью `ReactDOM.render`. Когда элемент будет создан, мы напишем утверждение:

```
test("renders a star", () => {
  const div = document.createElement("div");
  ReactDOM.render(<Star />, div);
  expect(div.querySelector("svg")).toBeTruthy();
});
```

Мы ожидаем, что, если мы попытаемся выбрать элемент `svg` внутри созданного `div`, результат будет правдивым. Когда мы запустим тест, должны увидеть, что код его прошел. Чтобы убедиться, что мы не получаем правильное утверждение, когда не должны, специально зададим неправильные значения и посмотрим, как тест завершится неудачей:

```
expect(
  div.querySelector("notrealthing")
).toBeTruthy();
```

В документации (oreil.ly/ah7ZU) есть более подробная информация обо всех доступных настраиваемых сопоставителях. Она поможет вам тестировать именно то, что вы хотите.

Когда вы создали свой проект React, вы могли заметить, что в дополнение к стандартным React и ReactDOM установилось несколько пакетов из @test-library. React Testing Library — это проект, начатый Кентом С. Доддсом как способ внедрения передовых методов тестирования и расширения утилит тестирования, которые были частью экосистемы React. Testing Library — это общее название множества пакетов тестирования не только React, но и таких библиотек, как Vue, Svelte, Reason, Angular и др.

Одна из причин, по которой вы можете выбрать React Testing Library, — это получение более подробных сообщений о непройденных тестах. Текущая ошибка, которую мы видим, когда проверяем утверждение:

```
expect(
  div.querySelector("notrealthing")
).toBeTruthy();
```

есть:

```
expect(received).toBeTruthy()

Received: null
```

Теперь добавим React Testing Library. Она уже установлена в проекте Create React App. Для начала импортируем функцию `toHaveAttribute` из @testing-library/jest-dom:

```
import { toHaveAttribute } from "@testing-library/jest-dom";
```

Теперь расширим функциональность `expext`, чтобы охватить и эту функцию:

```
expect.extend({ toHaveAttribute });
```

Вместо использования `toBeTruthy`, который дает трудночитаемые сообщения, применим следующее:

```
test("renders a star", () => {
  const div = document.createElement("div");
  ReactDOM.render(<Star />, div);
  expect(
    div.querySelector("svg")
  ).toHaveAttribute("id", "hotdog");
});
```

Теперь, когда мы запускаем тесты, мы видим ошибку с подробной информацией:

```
expect(element).toHaveAttribute("id", "hotdog")
// element.getAttribute("id") === "hotdog"

Expected the element to have attribute:
  id="hotdog"
Received:
  id="star"
```

Ее довольно просто исправить:

```
expect(div.querySelector("svg")).toHaveAttribute(
  "id",
  "star"
);
```

Включение нескольких пользовательских сопоставителей означает, что вам нужно будет их импортировать, расширять и использовать:

```
import {
  toHaveAttribute,
  toHaveClass
} from "@testing-library/jest-dom";

expect.extend({ toHaveAttribute, toHaveClass });

expect(you).toHaveClass("evenALittle");
```

Если вы обнаружите, что импортируете слишком много сопоставителей и их становится сложно перечислять и отслеживать, импортируйте библиотеку extend-expect:

```
import "@testing-library/jest-dom/extend-expect";

// Remove this --> expect.extend({ toHaveAttribute, toHaveClass });
```

Утверждения будут продолжать выполняться, как ожидалось. Еще один интересный факт о приложении Create React заключается в том, что в файле setupTests.js, который поставляется с CRA, есть строка, которая уже включает помощников extend-expect. Если вы посмотрите на папку src, то увидите, что setupTests.js содержит:

```
// jest-dom adds custom jest matchers for asserting on DOM nodes.
// allows you to do things like:
// expect(element).toHaveTextContent(/react/i)
// learn more: https://github.com/testing-library/jest-dom
import "@testing-library/jest-dom/extend-expect";
```

Поэтому, если вы используете приложение Create React, вам даже не нужно включать импорт в свои тестовые файлы.

Запросы

Запросы — еще одна функция библиотеки тестирования React, которая позволяет выполнять поиск по определенным критериям. Чтобы продемонстрировать использование запроса, настроим компонент `Star`, включив в него заголовок. Это позволит написать общий стиль теста — проверку соответствия на основе текста:

```
export default function Star({ selected = false }) {
  return (
    <>
      <h1>Great Star</h1>
      <FaStar
        id="star"
        color={selected ? "red" : "grey"}
      />
    </>
  );
}
```

На секунду задумаемся о том, что мы пытаемся проверить. Мы хотим, чтобы компонент отображался, или проверяем, содержит ли `h1` правильный текст. Разобраться в этом нам поможет функция `render`, которая является частью библиотеки тестирования React. Функция `render` заменит `ReactDOM.render()`, поэтому тест будет выглядеть немного иначе. Начнем с импорта этой функции из библиотеки тестирования React:

```
import { render } from "@testing-library/react";
```

Функция `render` будет принимать один аргумент — компонент или элемент, который мы хотим отобразить. Она возвращает объект запросов, который можно использовать для проверки значений в этом компоненте или элементе. Мы будем использовать запрос `getByText`, который найдет первый соответствующий узел для запроса и выдаст ошибку, если ни один элемент не совпадет. Чтобы вернуть список всех совпадающих узлов, используем `getAllBy` для возврата массива:

```
test("renders an h1", () => {
  const { getByText } = render(<Star />);
  const h1 = getByText(/Great Star/);
  expect(h1).toHaveTextContent("Great Star");
});
```

`getByText` находит элемент `h1` через переданное ему регулярное выражение. Затем мы используем сопоставление Jest `toHaveTextContent`, чтобы описать, какой текст должен находиться в контейнере `h1`.

Запустите тесты, и они пройдут. Если мы изменим текст, переданный в функцию `toHaveTextContent()`, тест не пройдет.

Тестирование событий

Важная часть написания тестов — это тестирование событий, которые являются частью компонентов. Протестируем компонент `Checkbox`, который мы создали в главе 7:

```
export function Checkbox() {
  const [checked, setChecked] = useReducer(
    checked => !checked,
    false
  );

  return (
    <>
      <label>
        {checked ? "checked" : "not checked"}
        <input
          type="checkbox"
          value={checked}
          onChange={setChecked}
        />
      </label>
    </>
  );
}
```

Этот компонент использует хук `useReducer` для переключения флажка. Наша цель — создать автоматический тест, который установит флажок и изменит значение `checked`: `false` (по умолчанию) на `true`. Написание теста для установки флажка также запустит `useReducer` и проверит хук.

Запустим тест:

```
import React from "react";

test("Selecting the checkbox should change the value of checked to true", () => {
  // .. write a test
});
```

Сначала выберем элемент, для которого мы хотим запустить событие. На какой элемент мы хотим щелкнуть при автоматическом тесте? Воспользуемся одним из запросов библиотеки тестирования, чтобы найти этот элемент. Поскольку у входных данных есть метка, мы можем использовать `getByLabelText()`:

```
import { render } from "@testing-library/react";
import { Checkbox } from "./Checkbox";

test("Selecting the checkbox should change the value of checked to true", () =>
{
  const { getByLabelText } = render(<Checkbox />);
});
```

Когда компонент отображается впервые, его текстовая метка — `not checked`, поэтому мы можем с помощью регулярного выражения найти совпадение со строкой:

```
test("Selecting the checkbox should change the value of checked to true", () => {
  const { getByLabelText } = render(<Checkbox />);
  const checkbox = getByLabelText(/not checked/);
});
```

Сейчас это регулярное выражение чувствительно к регистру, поэтому, если вы хотите найти какой-либо регистр, вы можете добавить `i` в конец текстовой метки. Используйте этот метод с осторожностью, в зависимости от того, насколько свободным вы хотите сделать выбор запроса:

```
const checkbox = getByLabelText(/not checked/i);
```

Теперь флажок установлен. Запустим событие (установить флажок) и напишем утверждение, чтобы убедиться, что для свойства `checked` установлено значение `true`, когда флажок установлен:

```
mport { render, fireEvent } from "@testing-library/react"

test("Selecting the checkbox should change the value of checked to true", () =>
{
  const { getByLabelText } = render(<Checkbox />);
  const checkbox = getByLabelText(/not checked/i);
  fireEvent.click(checkbox);
  expect(checkbox.checked).toEqual(true);
});
```

Мы также можем добавить обратный переключатель, снова запустив событие и проверив, что для свойства установлено значение `false` при переключении. Мы изменили название теста, чтобы оно точнее отражало его суть:

```
test("Selecting the checkbox should toggle its value", () => {
  const { getByLabelText } = render(<Checkbox />);
  const checkbox = getByLabelText(/not checked/i);
  fireEvent.click(checkbox);
  expect(checkbox.checked).toEqual(true);
  fireEvent.click(checkbox);
  expect(checkbox.checked).toEqual(false);
});
```

В этом случае установить флажок довольно просто. У нас есть метка, которую мы можем использовать, чтобы найти входные данные, которые мы хотим проверить. Если у вас нет такого простого способа доступа к элементу DOM, в Testing Library есть другая утилита для проверки с любым элементом DOM. Начнем с добавления атрибута к элементу, который мы хотим выбрать:

```
<input
  type="checkbox"
  value={checked}
  onChange={setChecked}
  data-testid="checkbox" // Add the data-testid= attribute
/>
```

Затем используем запрос `getByTestId`:

```
test("Selecting the checkbox should change the value of checked to true", () => {
  const { getByTestId } = render(<Checkbox />);
  const checkbox = getByTestId("checkbox");
  fireEvent.click(checkbox);
  expect(checkbox.checked).toEqual(true);
});
```

Код будет делать то же самое, но особенно полезно это будет при обращении к элементам DOM, к которым трудно подобраться иным способом.

После тестирования этого компонента `Checkbox` мы можем уверенно включить его в остальную часть приложения и использовать повторно.

Использование понятия покрытия кода

Покрытие кода — это информация о том, сколько строк кода было фактически протестировано. Это показатель, который поможет решить, достаточно ли тестов мы написали.

В комплекте с Jest есть инструмент JavaScript Istanbul, который используется для просмотра тестов и создания отчета, в котором описывается, сколько операторов, ветвей, функций и строк охвачено тестами.

Чтобы запустить Jest с покрытием кода, просто добавьте флаг `coverage` при запуске команды `jest`:

```
npm test -- --coverage
```

Этот отчет сообщает, какая часть кода в каждом файле была выполнена в процессе тестирования, а также выдает перечень файлов, которые были импортированы в тесты.

Jest также создает отчет, который вы можете запустить в браузере. В отчете содержится более подробная информация о том, какой объем кода был охвачен тестами. После запуска Jest с отчетом о покрытии кода вы заметите, что в корень добавилась папка coverage. Откройте в веб-браузере файл /coverage/lcov-report/index.html. Он покажет покрытие кода в интерактивном отчете.

Этот отчет сообщает, какая часть кода была охвачена тестами, а также показывает индивидуальное покрытие каждой подпапки. Вы можете зайти в подпапку, чтобы увидеть, насколько хорошо были покрыты отдельные файлы в ней. Если вы выберете папку components/ui, то увидите, насколько хорошо компоненты пользовательского интерфейса охватываются тестированием.

Чтобы посмотреть, какие строки охвачены тестами в отдельном файле, щелкните имя файла.

Покрытие кода — отличный инструмент для измерения охвата тестов. Это один из способов понять, написали ли вы достаточно юнит-тестов для своего кода. 100 % покрытие кода достигается нечасто. Стоит ориентироваться на планку выше 85 %[1].

Тестирование часто может казаться излишним, но инструменты тестирования React никогда не были лучше. Даже если вы не тестируете весь код, само размышление о том, как внедрить методы тестирования, может помочь сэкономить время и деньги при создании готовых к работе приложений.

[1] См. статью Мартина Фаулера «Test-Coverage» (oreil.ly/Hbb-D)

ГЛАВА 11

React Router

На заре интернета большинство сайтов состояло из серии страниц, по которым пользователи могли перемещаться путем запросов и открытия отдельных файлов. Местоположение текущего файла или ресурса отображалось в адресной строке браузера. Кнопки перемещения вперед и назад работали вполне ожидаемым образом. Содержимое закладок, нацеленных на глубины сайта, позволяло пользователям сохранять ссылки на конкретный файл, который мог быть заново загружен по запросу пользователя. На сайте со страничной организацией, или с серверным представлением, браузерная навигация и истории просмотров просто работали, как и планировалось.

В одностраничном приложении (single-page application, SPA) описанные функции стали проблематичными. Напоминаем, что в данном приложении все происходит на одной и той же странице. Средства языка JavaScript загружают информацию и вносят изменения в пользовательский интерфейс. Такие свойства, как история просмотров, закладки и кнопки перемещений вперед и назад, не будут работать без решений, связанных с *маршрутизацией*. Она представляет собой процесс определения конечных точек запросов пользователей[1]. Эти точки работают в связке с местоположением и объектами истории просмотров браузера. Они используются для идентификации запрошенного содержимого, чтобы средства JavaScript могли загрузить и вывести на экран соответствующий пользовательский интерфейс.

В отличие от Angular, Ember или Backbone, React не поставляется со стандартным маршрутизатором. Осознавая важность решений по маршрутизации, инженеры Майкл Джексон (Michael Jackson) и Райан Флоренс (Ryan Florence) создали маршрутизатор, который назвали просто React Router. Он был благосклонно воспринят сообществом в качестве популярного решения по марш-

[1] Express.js documentation: Basic Routing. https://expressjs.com/en/starter/basic-routing.html.

рутизации для приложений React[1]. Его используют такие компании, как Uber, Zendesk, PayPal и Vimeo[2].

В этой главе мы познакомимся с React Router и воспользуемся его функциями для обработки маршрутизации на стороне клиента.

Внедрение Router

Чтобы продемонстрировать возможности React Router, создадим классический стартовый сайт с разделами «О нас», «События», «Продукты» и «Контакты». Хотя этот сайт выглядит так, словно состоит из нескольких страниц, на самом деле это одностраничное приложение (рис. 11.1).

Рис. 11.1. Простой сайт с навигацией по ссылкам

Карта сайта для этого веб-сайта состоит из домашней страницы, страницы для каждого раздела и страницы ошибок для обработки ошибок 404 Not Found (рис. 11.2).

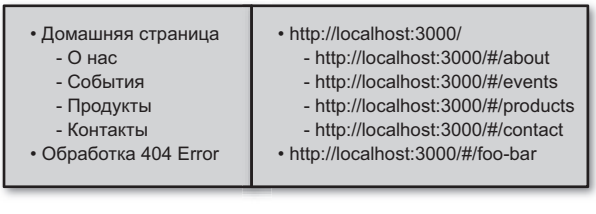


Рис. 11.2. Карта сайта с локальными ссылками

[1] Этот отчет был отмечен на GitHub звездочкой более 20 000 раз. https://github.com/ReactTraining/react-router/stargazers

[2] См. сайты, использующие React Router, oreil.ly/staEF.

Маршрутизатор позволяет настраивать маршруты для каждого раздела сайта. Каждый маршрут — это конечная точка, которую можно ввести в адресную строку браузера. Когда запрашивается маршрут, мы можем отобразить соответствующий контент.

Для начала установим React Router и React Router DOM. React Router DOM применяется для обычных приложений React, которые используют DOM. Если вы пишете приложение для React Native, вы будете использовать response-router-native. Мы установим эти пакеты в экспериментальных версиях, потому что React Router 6 официально на момент публикации не вышел. После его выпуска вы сможете использовать пакеты без этого обозначения:

```
npm install react-router@experimental react-router-dom@experimental
```

Также понадобится несколько компонентов-заполнителей для каждого раздела или страницы в карте сайта. Экспортируем их из файла pages.js:

```
import React from "react";

export function Home() {
  return (
    <div>
      <h1>[Company Website]</h1>
    </div>
  );
}

export function About() {
  return (
    <div>
      <h1>[About]</h1>
    </div>
  );
}

export function Events() {
  return (
    <div>
      <h1>[Events]</h1>
    </div>
  );
}

export function Products() {
  return (
    <div>
      <h1>[Products]</h1>
    </div>
  );
```

```
}

export function Contact() {
  return (
    <div>
      <h1>[Contact]</h1>
    </div>
  );
}
```

Когда заготовки для страниц готовы, настраиваем файл index.js. Вместо рендеринга компонента `App` мы рендерим компонент `Router`. Компонент `Router` передает информацию о текущем местоположении всем дочерним элементам, вложенным в него. Его следует использовать один раз и поместить рядом с корнем дерева компонентов:

```
import React from "react";
import { render } from "react-dom";
import App from "./App";

import { BrowserRouter as Router } from "react-router-dom";

render(
  <Router>
    <App />
  </Router>,
  document.getElementById("root")
);
```

Обратите внимание, что мы импортируем объект `BrowserRouter` как `Router`. Следующее, что нам нужно сделать, это настроить конфигурацию маршрута. Поместим ее в файл App.js. Компонент-оболочка для любых маршрутов, которые мы хотим отобразить, называется `Routes`. Внутри `Routes` мы будем использовать компонент `Route` для каждой страницы, которую хотим отобразить. Мы также хотим импортировать все страницы из файла ./pages.js:

```
import React from "react";
import { Routes, Route } from "react-router-dom";
import {
  Home,
  About,
  Events,
  Products,
  Contact
} from "./pages";

function App() {
  return (
    <div>
```

```
      <Routes>
        <Route path="/" element={<Home />} />
        <Route
          path="/about"
          element={<About />}
        />
        <Route
          path="/events"
          element={<Events />}
        />
        <Route
          path="/products"
          element={<Products />}
        />
        <Route
          path="/contact"
          element={<Contact />}
        />
      </Routes>
    </div>
  );
}
```

Эти маршруты сообщают `Router`, какой компонент отображать при изменении положения окна. Каждый компонент `Route` имеет свойства `path` и `element`. Когда расположение браузера совпадает с `path`, отобразится `element`. Если местоположение равно `/`, маршрутизатор отобразит компонент `Home`. Если расположение является `/products`, маршрутизатор отобразит компонент `Products`.

На этом этапе мы можем запустить приложение и физически ввести маршруты в адресную строку браузера, следя за изменением содержимого. Например, введем в адресную строку http://localhost:3000/about и посмотрим, как отобразится компонент «О программе».

Не стоит ожидать, что пользователи будут перемещаться по веб-сайту, вводя в адресную строку маршруты. У react-router-dom есть компонент `Link`, который мы можем использовать для создания ссылок для браузера.

Изменим домашнюю страницу, чтобы она содержала меню навигации со ссылкой на каждый маршрут:

```
import { Link } from "react-router-dom";

export function Home() {
  return (
    <div>
      <h1>[Company Website]</h1>
      <nav>
        <Link to="about">About</Link>
```

```
        <Link to="events">Events</Link>
        <Link to="products">Products</Link>
        <Link to="contact">Contact Us</Link>
      </nav>
    </div>
  );
}
```

Теперь пользователи могут получать доступ к каждой внутренней странице с домашней страницы, щелкнув по ссылке. Кнопка «Назад» в браузере вернет их на главную страницу.

Свойства маршрутизатора

React Router передает свойства компонентам, которые он отображает. Например, мы можем получить текущее местоположение через свойство. Используем текущее местоположение, чтобы создать компонент 404 Not Found. Сначала создадим компонент:

```
export function Whoops404() {
  return (
    <div>
      <h1>Resource not found</h1>
    </div>
  );
}
```

Затем добавим его в конфигурацию маршрута в файле App.js. Если мы вводим несуществующий маршрут, например `highway`, нужно отобразить компонент `Whoops404`. Мы будем использовать * как значение пути, а компонент — как элемент:

```
function App() {
  return (
    <div>
      <Routes>
        <Route path="/" element={<Home />} />
        <Route
          path="/about"
          element={<About />}
        />
        <Route
          path="/events"
          element={<Events />}
        />
        <Route
          path="/products"
          element={<Products />}
```

```
        />
        <Route
          path="/contact"
          element={<Contact />}
        />
        <Route path="*" element={<Whoops404 />} />
      </Routes>
    </div>
  );
}
```

Теперь, если мы введем ссылку localhost:3000/highway, то увидим компонент страницы 404. Мы также можем отображать значение маршрута, который мы посетили, используя значение местоположения. Поскольку мы живем в мире, где есть хуки React, используем хук. В компоненте `Whoops404` создадим переменную с именем `location`, которая возвращает значение текущего местоположения (то есть свойство с информацией о том, на какую страницу вы перешли). Затем используем значение `location.pathname` для рендеринга посещаемого маршрута:

```
export function Whoops404() {
  let location = useLocation();
  console.log(location);
  return (
    <div>
      <h1>
        Resource not found at {location.pathname}
      </h1>
    </div>
  );
}
```

Если мы введем в журнал `location`, то сможем исследовать этот объект дальше.

В этом разделе мы ввели основы реализации и работы React Router. `Router` используется один раз и включает в себя все компоненты, которые будут использовать маршрутизацию. Все компоненты `Route` должны быть обернуты компонентом `Routes`, который выбирает компонент для рендеринга на основе текущего местоположения окна. Компоненты `Link` могут использоваться для облегчения навигации. Эти основы важны, но они лишь поверхностно раскрывают возможности маршрутизатора.

Вложенность маршрутов

Компоненты маршрута используются с контентом, который должен отображаться только при совпадении определенных URL-адресов. Эта функция позволяет организовывать веб-приложения в информативные иерархии, которые способствуют повторному использованию контента.

Иногда, когда пользователи перемещаются по приложениям, мы хотим, чтобы часть пользовательского интерфейса оставалась на месте. В прошлом такие решения, как шаблоны страниц и главные страницы, помогали веб-разработчикам повторно использовать элементы пользовательского интерфейса.

Рассмотрим простой стартовый сайт. Возможно, мы захотим создать подстраницы для страницы «О нас», на которой будет отображаться дополнительный контент. Когда пользователь выберет раздел «О нас», по умолчанию он перейдет на страницу «Компания» в этом разделе. Схема сайта выглядит так:

- Домашняя страница
  - **О нас**
    - **Компания (по умолчанию)**
    - **История**
    - **Сервисы**
    - **Карта**
  - События
  - Продукты
  - Контакты
- Страница ошибки 404

Новые маршруты, которые нужно создать, будут отражать эту иерархию:

- http://localhost:3000/
  - http://localhost:3000/about
    - http://localhost:3000/about
    - http://localhost:3000/about/history
    - http://localhost:3000/about/services
    - http://localhost:3000/about/location
  - http://localhost:3000/events
  - http://localhost:3000/products
  - http://localhost:3000/contact
- http://localhost:3000/hot-potato

Также нужно не забыть вставить компоненты-заполнители для новых разделов: «Компания», «Услуги», «История» и «Карта». В качестве примера приведем текст для компонента `Services`, который вы можете повторно использовать для двух других компонентов:

```
export function Services() {
  <section>
    <h2>Our Services</h2>
    <p>
      Lorem ipsum dolor sit amet, consectetur
      adipiscing elit. Integer nec odio. Praesent
      libero. Sed cursus ante dapibus diam. Sed
      nisi. Nulla quis sem at nibh elementum
      imperdiet. Duis sagittis ipsum. Praesent
      mauris. Fusce nec tellus sed augue semper
      porta. Mauris massa. Vestibulum lacinia arcu
      eget nulla. Class aptent taciti sociosqu ad
      litora torquent per conubia nostra, per
      inceptos himenaeos. Curabitur sodales ligula
      in libero.
    </p>
  </section>;
}
```

Создав эти компоненты, мы можем настроить маршрутизатор, начиная с файла App.js. Если вы хотите создать иерархию страниц с маршрутами, все, что вам нужно сделать, это вложить компоненты `Route` друг в друга:

```
import {
  Home,
  About,
  Events,
  Products,
  Contact,
  Whoops404,
  Services,
  History,
  Location
} from "./pages";

function App() {
  return (
    <div>
      <Routes>
        <Route path="/" element={<Home />} />
        <Route path="about" element={<About />}>
          <Route
            path="services"
            element={<Services />}
          />

          <Route
            path="history"
            element={<History />}
          />
          <Route
```

```
            path="location"
            element={<Location />}
          />
        </Route>
        <Route
          path="events"
          element={<Events />}
        />
        <Route
          path="products"
          element={<Products />}
        />
        <Route
          path="contact"
          element={<Contact />}
        />
        <Route path="*" element={<Whoops404 />} />
      </Routes>
    </div>
  );
}
```

После того как вы обернули вложенные маршруты в компонент `Route`, вы можете посетить страницу. Открыв ссылку http://localhost:3000/about/history, вы увидите только содержимое страницы `About` но компонент `History` не отобразится. Чтобы его отобразить, используем компонент `Outlet`. Он позволит визуализировать эти вложенные компоненты. Мы просто разместим его в любом месте, где хотим отобразить дочерний контент.

В компоненте `About` в pages.js добавим `Outlet` в `<h1>`:

```
import {
  Link,
  useLocation,
  Outlet
} from "react-router-dom";

export function About() {
  return (
    <div>
      <h1>[About]</h1>
      <Outlet />
    </div>
  );
}
```

Теперь компонент `About` будет многократно использоваться во всем разделе и отображать вложенные компоненты. Адрес сообщит приложению, какой подраздел нужно отобразить. Например, при расположении http://localhost:3000/about/history компонент `History` будет отображаться внутри компонента `About`.

Использование переадресации

Иногда нужно перенаправить пользователей с одного маршрута на другой. Например, мы хотим убедиться, что если пользователи попытаются получить доступ к контенту через http://localhost:3000/services, они будут перенаправлены на верный маршрут: http://localhost:3000/about/services.

Добавим перенаправления, чтобы пользователи могли получить доступ к правильному контенту:

```
import {
  Routes,
  Route,
  Redirect
} from "react-router-dom";

function App() {
  return (
    <div>
      <Routes>
        <Route path="/" element={<Home />} />
        // Other Routes
        <Redirect
          from="services"
          to="about/services"
        />
      </Routes>
    </div>
  );
}
```

Компонент `Redirect` позволяет перенаправить пользователя на определенный маршрут.

При изменении маршрутов в продакшене пользователи все равно будут пытаться получить доступ к старому контенту по старым маршрутам. Обычно это происходит из-за закладок. Компонент `Redirect` дает возможность загружать правильный контент, даже если пользователь обращается к сайту через старую закладку.

В этом разделе мы создали конфигурацию маршрута с помощью компонента `Route`. Если вам нравится эта структура, можете пропустить следующий раздел, но мы хотели рассказать, как создать конфигурацию маршрута по-другому. Также для настройки маршрутизации можно использовать перехватчик `useRoutes`.

Чтобы переделать приложение под использование `useRoutes`, нужно внести изменения в компонент `App` (или туда, где настроены маршруты). Реорганизуем его:

```
import { useRoutes } from "react-router-dom";

function App() {
  let element = useRoutes([
    { path: "/", element: <Home /> },
    {
      path: "about",
      element: <About />,
      children: [
        {
          path: "services",
          element: <Services />
        },
        { path: "history", element: <History /> },
        {
          path: "location",
          element: <Location />
        }
      ]
    },
    { path: "events", element: <Events /> },
    { path: "products", element: <Products /> },
    { path: "contact", element: <Contact /> },
    { path: "*", element: <Whoops404 /> },
    {
      path: "services",
      redirectTo: "about/services"
    }
  ]);
  return element;
}
```

Официальная документация гласит, что настройку нужно назвать `element`, но вы можете называть ее как хотите. Также совершенно необязательно использовать этот синтаксис. `Route` — это оболочка для `useRoutes`, так что у вас в любом случае все получится. Выберите тот синтаксис и стиль, который вам больше подходит!

Параметры маршрутизации

Еще одна полезная функция React Router — это возможность настраивать параметры маршрутизации — переменные, которые получают значения из URL-адреса. Они чрезвычайно полезны в управляемых данными веб-приложениях для фильтрации контента или управления настройками рендеринга.

Вернемся к органайзеру цветов и улучшим его, добавив возможность выбирать и отображать цвета по одному с помощью React Router. Когда пользователь выбирает цвет, щелкая по нему, приложение должно отобразить этот цвет, его заголовок и шестнадцатеричное значение.

Используя маршрутизатор, мы можем получить идентификатор цвета через URL-адрес. Например, вот такой URL-адрес мы будем использовать для рендеринга цвета «зеленый газон», потому что его ID передается в URL-адрес:

```
http://localhost:3000/58d9caee-6ea6-4d7b-9984-65b145031979
```

Для начала настроим маршрутизатор в файле index.js. Импортируем `Router` и обернем компонент `App`:

```
import { BrowserRouter as Router } from "react-router-dom";

render(
  <Router>
    <App />
  </Router>,
  document.getElementById("root")
);
```

При упаковке `App` все свойства маршрутизатора передаются компоненту и другим компонентам, вложенным в него. После этого мы можем настроить конфигурацию маршрута. Мы используем компоненты `Routes` и `Route` вместо `useRoutes`, но помните, что можно выбрать другой синтаксис. Начнем с импорта `Routes` и `Route`:

```
import { Routes, Route } from "react-router-dom";
```

Затем добавим их в `App`. У этого приложения будет два маршрута: `ColorList` и `ColorDetails`. Мы не создали `ColorDetailsyet`, поэтому импортируем его:

```
import { ColorDetails } from "./ColorDetails";

export default function App() {
  return (
    <ColorProvider>
      <AddColorForm />
      <Routes>
        <Route
          path="/"
          element={<ColorList />}
        />
        <Route
          path=":id"
          element={<ColorDetails />}
        />
      </Routes>
    </ColorProvider>
  );
}
```

Компонент `ColorDetails` будет отображаться динамически в зависимости от `id` цвета. Создадим компонент `ColorDetails` в новом файле с именем ColorDetails.js. Для начала это будет пустая заготовка:

```
import React from "react";

export function ColorDetails() {
  return (
    <div>
      <h1>Details</h1>
    </div>
  );
}
```

Как понять, работает ли этот код? Самый простой способ проверить — открыть инструменты разработчика React и найти идентификатор одного из отображаемых цветов. Если у вас еще нет цвета, добавьте его и посмотрите его идентификатор. Получив идентификатор, вы сможете добавить его к URL-адресу `localhost:3000`. Например, `localhost:3000/00fdb4c5-c5bd-4087-a48f-4ff7a9d90af8`.

Вы увидите страницу ColorDetails. Теперь мы знаем, что маршрутизатор и наши маршруты работают, но хотим сделать эту работу более динамичной. На странице ColorDetails мы хотим отображать правильный цвет по `id` из URL-адреса. Для этого воспользуемся хуком `useParams`:

```
import { useParams } from "react-router-dom";

export function ColorDetails() {
  let params = useParams();
  console.log(params);
  return (
    <div>
      <h1>Details</h1>
    </div>
  );
}
```

Если мы введем `params` в консоль, то увидим, что это объект, который содержит любые параметры маршрутизатора. Мы деструктурируем этот объект, чтобы получить идентификатор и использовать его для поиска правильного цвета в массиве `colors`. Используем хук `useColors`, чтобы реализовать это:

```
import { useColors } from "./";

export function ColorDetails() {
  let { id } = useParams(); // destructure id
  let { colors } = useColors();

  let foundColor = colors.find(
```

```
    color => color.id === id
  );
  console.log(foundColor);

  return (
    <div>
      <h1>Details</h1>
    </div>
  );
}
```

Введя в консоль `foundColor`, мы увидим, что нашли правильный цвет. Теперь нам нужно только отобразить данные об этом цвете в компоненте:

```
export function ColorDetails() {
  let { id } = useParams();
  let { colors } = useColors();

  let foundColor = colors.find(
    color => color.id === id
  );

  return (
    <div>
      <div
        style={{
          backgroundColor: foundColor.color,
          height: 100,
          width: 100
        }}
      ></div>
      <h1>{foundColor.title}</h1>
      <h1>{foundColor.color}</h1>
    </div>
  );
}
```

Еще одна функция, которую мы хотим добавить в органайзер цветов, — это возможность переходить на страницу `ColorDetails` щелчком по цвету в списке. Добавим эту функциональность в компонент `Color`. Используем другой хук — `useNavigate`, который будет открывать страницу сведений, когда мы нажимаем на компонент. Сначала импортируем его из response-router-dom:

```
import { useNavigate } from "react-router-dom";
```

Затем вызовем функцию `useNavigate`, которая вернет функцию для перехода на другую страницу:

```
let navigate = useNavigate();
```

Добавим в этом разделе обработчик `onClick` для перехода к маршруту на основе идентификатора цвета:

```
let navigate = useNavigate();

return (
  <section
    className="color"
    onClick={() => navigate(`/${id}`)}
  >
    // Color component
  </section>
);
```

Теперь, если мы щелкнем по `section`, то перейдем на правильную страницу.

Параметры маршрутизации — идеальный инструмент для получения данных, влияющих на представление пользовательского интерфейса. Однако их следует использовать только в том случае, если вы хотите, чтобы пользователи фиксировали эти данные в URL-адресе. В случае органайзера цветов пользователи могут отправлять другим пользователям ссылки на определенные цвета или все цвета, отсортированные по определенному полю, или добавлять эти ссылки в закладки, чтобы возвращать определенные данные.

В этой главе мы рассмотрели базовое использование React Router. В следующей главе мы узнаем, как использовать маршрутизацию на сервере.

ГЛАВА 12

React и сервер

До сих пор мы создавали небольшие приложения с React, запускаемые исключительно в браузере. Они собирали данные в браузере, которые содержались в его хранилище. В этом был вполне определенный смысл, поскольку React относится к уровню представления данных. Его предназначением является отображение пользовательского интерфейса. И тем не менее для многих приложений требуется серверная часть программы, и нам нужно понимать, как создавать структуру приложений с учетом применения сервера.

Даже если клиентское приложение целиком зависит от облачных сервисов, вам все равно понадобится получать и отправлять данные, используя эти сервисы. В архитектуре Flux есть конкретные места, где должны совершаться транзакции, а также библиотеки, способные справиться с задержками, связанными с HTTP-запросами.

Кроме того, React способна на *изоморфные* режимы отображения, то есть может применяться на платформах, отличных от браузера. Следовательно, пользовательский интерфейс можно отображать на сервере перед тем, как он попадет в браузер. Серверное отображение может повысить производительность, надежность и безопасность приложений.

В начале главы мы рассмотрим различия между изоморфизмом и универсализмом, а также отметим, как обе эти концепции относятся к React. Далее мы рассмотрим, как создать изоморфное приложение, используя универсальный JavaScript. Наконец, мы улучшим наше приложение рецептов, добавив сервер и отобразив пользовательский интерфейс на сервере.

Изоморфность против универсальности

Термины *изоморфный* и *универсальный* часто используются как синонимы для описания приложений, которые работают как на клиенте, так и на сервере.

Однако между ними есть тонкое различие, которое стоит изучить. *Изоморфные* приложения могут отображаться на разных платформах. *Универсальный* код может работать в нескольких средах[1].

Node.js позволяет повторно использовать тот же код, который мы написали в браузере, в других приложениях, например серверах, интерфейсах командной строки и даже собственных приложениях. Посмотрим на универсальный JavaScript:

```
const userDetails = response => {
  const login = response.login;
  console.log(login);
};
```

Функция `printNames` универсальная. Такой код можно вызвать как в браузере, так и на сервере. Это означает, что если мы построим сервер с Node.js, то потенциально сможем повторно использовать код между двумя средами. Универсальный JavaScript — это JavaScript, который может работать на сервере или в браузере без ошибок (рис. 12.1).

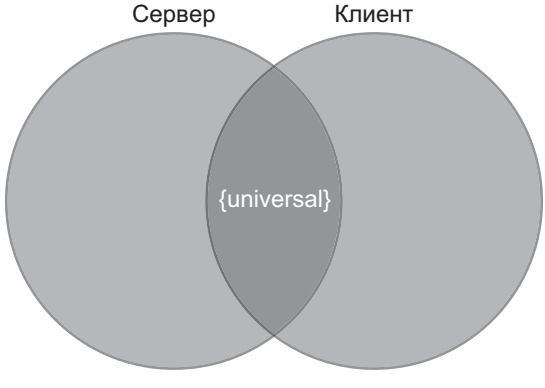


Рис. 12.1. Клиентские и серверные среды

Клиентские и серверные среды

Сервер и клиент — это совершенно разные вещи, поэтому весь наш код JavaScript не будет автоматически работать и там, и там. Посмотрим, как создать запрос AJAX в браузере:

[1] Хенгевельд Герт. Isomorphism vs Universal JavaScript, Medium (oreil.ly/i70W2)

```
fetch("https://api.github.com/users/moonhighway")
  .then(res => res.json())
  .then(console.log);
```

Здесь мы делаем запрос на выборку в GitHub API, конвертируем ответ в JSON, а затем вызываем функцию с результатами JSON для его анализа.

Однако, если мы попытаемся запустить тот же код с Node.js, то получим ошибку:

```
fetch("https://api.github.com/users/moonhighway")
^

ReferenceError: fetch is not defined
at Object.<anonymous> (/Users/eveporcello/Desktop/index.js:7:1)
at Module.\_compile (internal/modules/cjs/loader.js:1063:30)
at Object.Module.\_extensions..js (internal/modules/cjs/loader.js:1103:10)
at Module.load (internal/modules/cjs/loader.js:914:32)
at Function.Module.\_load (internal/modules/cjs/loader.js:822:14)
at Function.Module.runMain (internal/modules/cjs/loader.js:1143:12)
at internal/main/run_main_module.js:16:11
```

Эта ошибка возникает из-за того, что в Node.js нет встроенной функции `fetch`, как в браузере. В Node.js мы можем использовать функцию `isomorphic-fetch` из npm или встроенный модуль `https`. Поскольку мы уже использовали синтаксис `fetch`, добавим `isomorphic-fetch`:

```
npm install isomorphic-fetch
```

Затем мы просто импортируем `isomorphic-fetch` без изменений в коде:

```
const fetch = require("isomorphic-fetch");

const userDetails = response => {
  const login = response.login;
  console.log(login);
};

fetch("https://api.github.com/users/moonhighway")
  .then(res => res.json())
  .then(userDetails);
```

Загрузка данных из API с помощью Node.js требует использования основных модулей. Нужен другой код. В этих примерах функция `userDetails` универсальная, то есть работает в обеих средах.

Этот файл JavaScript теперь изоморфный, поскольку он содержит универсальный JavaScript. Не весь код здесь универсальный, но сам файл будет работать

в обеих средах. Его можно запустить с помощью Node.js или включить в тег `<script>` в браузере.

Посмотрим на компонент `Star`. Универсальный ли он?

```
function Star({
  selected = false,
  onClick = f => f
}) {
  return (
    <div
      className={
        selected ? "star selected" : "star"
      }
      onClick={onClick}
    ></div>
  );
}
```

Конечно, да, ведь JSX компилируется в JavaScript. Компонент `Star` — это просто функция:

```
function Star({
  selected = false,
  onClick = f => f
}) {
  return React.createElement("div", {
    className: selected
      ? "star selected"
      : "star",
    onClick: onClick
  });
}
```

Мы можем отобразить этот компонент непосредственно в браузере или другой среде и записать HTML-вывод в виде строки. В `ReactDOM` есть метод `renderToString` для рендеринга пользовательского интерфейса в строку HTML:

```
// Renders html directly in the browser
ReactDOM.render(<Star />);

// Renders html as a string
let html = ReactDOM.renderToString(<Star />);
```

Мы можем создавать изоморфные приложения, которые отображают компоненты на разных платформах, и проектировать эти приложения таким образом, чтобы повторно использовать код JavaScript универсально в разных средах. Кроме того, мы можем создавать изоморфные приложения, используя другие языки, такие как Go или Python, — мы не ограничены Node.js.

Рендеринг React на сервере

Метод `ReactDOM.renderToString` позволяет отображать пользовательский интерфейс на сервере. Серверы мощные, и у них есть доступ ко всем видам ресурсов, которого нет у браузеров. Серверы могут быть безопаснее и иметь доступ к защищенным данным. Отображая контент на сервере, вы можете использовать все эти преимущества.

Попробуем отобразить на сервере наше приложение рецептов, которое мы создали в главе 5. Запустим Create React App и поместим этот код поверх содержимого файла index.js:

```
import React from "react";
import ReactDOM from "react-dom";
import "./index.css";
import { Menu } from "./Menu";

const data = [
  {
    name: "Baked Salmon",
    ingredients: [
      {
        name: "Salmon",
        amount: 1,
        measurement: "lb"
      },
      {
        name: "Pine Nuts",
        amount: 1,
        measurement: "cup"
      },
      {
        name: "Butter Lettuce",
        amount: 2,
        measurement: "cups"
      },
      {
        name: "Yellow Squash",
        amount: 1,
        measurement: "med"
      },
      {
        name: "Olive Oil",
        amount: 0.5,
        measurement: "cup"
      },
      {
        name: "Garlic",
        amount: 3,
```

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

ReactDOM.render(
```

```
  <Menu
    recipes={data}
    title="Delicious Recipes"
  />,
  document.getElementById("root")
);
```

Компоненты расположены в новом файле с именем Menu.js:

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

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

На протяжении всей книги мы отображали компоненты на клиенте. Рендеринг на стороне клиента — это первое, что мы делаем при создании приложения. Мы обслуживаем папку build Create React App, а браузер запускает HTML и вызывает файл script.js для загрузки любого JavaScript.

Это может занять много времени. В зависимости от скорости сети пользователю, возможно, придется подождать несколько секунд, чтобы увидеть что-нибудь. Используя Create React App с сервером Express, мы можем создать гибрид рендеринга на стороне клиента и сервера.

Мы отобразим компонент `Menu`, который выводит на экран несколько рецептов. Первое изменение, которое мы внесем в это приложение, — заменим `ReactDOM.render` на `ReactDOM.hydrate`.

Эти две функции идентичны, за исключением того, что `hydrate` используется для добавления содержимого в контейнер, который был отображен `ReactDOMServer`. Порядок действий такой:

1. Отображаем статическую версию приложения, позволяющую пользователям видеть, что что-то происходит и страница «загружается».
2. Делаем запрос на динамический JavaScript.
3. Заменяем статический контент на динамический.
4. Пользователь нажимает на что-то, и это работает.

Мы повторно запускаем приложение после рендеринга на стороне сервера. Под этим мы подразумеваем статическую загрузку содержимого в виде статического HTML и затем загрузку JavaScript. Это позволит пользователям ощутить производительность. Они увидят, что на странице что-то происходит, и это заставит их захотеть остаться на странице.

В качестве сервера проекта мы используем Express — легкий сервер Node. Сначала установим его:

```
npm install express
```

Затем создадим папку сервера с именем server и внутри нее создадим файл index.js. Этот файл создаст сервер, который будет обслуживать папку build, но также предварительно загрузит статический HTML-контент:

```
import express from "express";
const app = express();

app.use(express.static("./build"));
```

Этот код импортирует и статически обслуживает папку build. Затем мы используем функцию `renderToString` из ReactDOM для рендеринга приложения как статической HTML-строки:

```
import React from "react";
import ReactDOMServer from "react-dom/server";
```

```
import { Menu } from "../src/Menu.js";

const PORT = process.env.PORT || 4000;

app.get("/*", (req, res) => {
  const app = ReactDOMServer.renderToString(
    <Menu />
  );
});

app.listen(PORT, () =>
  console.log(
    `Server is listening on port ${PORT}`
  )
);
```

Мы передадим компонент `Menu` в эту функцию, потому что именно его мы хотим отобразить статически. Затем прочитаем статический файл index.html из созданного клиентского приложения, вставим содержимое приложения в `div` и отправим его в качестве ответа на запрос:

```
app.get("/*", (req, res) => {
  const app = ReactDOMServer.renderToString(
    <Menu />
  );

  const indexFile = path.resolve(
    "./build/index.html"
  );

  fs.readFile(indexFile, "utf8", (err, data) => {
    return res.send(
      data.replace(
        '<div id="root"></div>',
        `<div id="root">${app}</div>`
      )
    );
  });
});
```

Далее выполним настройку с помощью webpack и Babel. Помните, что Create React App по умолчанию умеет выполнять компиляцию и сборку, но в серверном проекте нужно настроить и обеспечить соблюдение различных правил.

Начнем с установки парочки (ну или больше) зависимостей:

```
npm install @babel/core @babel/preset-env babel-loader nodemon npm-run-all
webpack webpack-cli webpack-node-externals
```

После установки Babel создадим `.babelrc` с некоторыми предустановками:

```
{
  "presets": ["@babel/preset-env", "react-app"]
}
```

Добавим react-app, потому что в проекте используется Create React App.

Затем добавим файл конфигурации webpack.server.js:

```
const path = require("path");
const nodeExternals = require("webpack-node-externals");

module.exports = {
  entry: "./server/index.js",
  target: "node",
  externals: [nodeExternals()],
  output: {
    path: path.resolve("build-server"),
    filename: "index.js"
  },
  module: {
    rules: [
      {
        test: /\.js$/,
        use: "babel-loader"
      }
    ]
  }
};
```

`babel-loader` преобразует файлы JavaScript, как и ожидалось, а `nodeExternals` сканирует папку node_modules на предмет всех имен `node_modules`. Затем он создает внешнюю функцию, которая приказывает webpack не связывать эти модули или подмодули.

Мы можем столкнуться с ошибкой веб-пакета из-за конфликта между версией, которую мы установили с помощью Create React App, и версией, установленной нами только что. Чтобы исправить конфликт, просто добавим файл .env в корень проекта и этот код:

```
SKIP_PREFLIGHT_CHECK=true
```

Наконец, мы можем добавить несколько дополнительных сценариев npm для запуска команд разработчика:

```
{
  "scripts": {
    //...
    "dev:build-server": "NODE_ENV=development webpack --config webpack.server.js
    --mode=development -w",
```

```
    "dev:start": "nodemon ./server-build/index.js",
    "dev": "npm-run-all --parallel build dev:*"
  }
}
```

1. `dev:build-server` передает `development` в качестве переменной среды и запускает `webpack` с новой конфигурацией сервера.
2. `dev:start` запускает файл сервера с помощью `nodemon`, который будет отслеживать любые изменения.
3. `dev` запускает оба процесса параллельно.

Теперь, когда мы запускаем команду `npm run dev`, оба процесса будут работать. Мы должны увидеть, как приложение работает на localhost:4000. Когда приложение запускается, содержимое загружается последовательно, сначала в виде предварительно отображенного HTML, а затем с помощью пакета JavaScript.

Использование подобной техники может ускорить загрузку и повысить воспринимаемую производительность. Поскольку пользователи ожидают, что время загрузки страницы составит две секунды или меньше, любое повышение производительности может переманить пользователя от одного конкурентного приложения к другому.

Серверный рендеринг с Next.js

Еще один мощный и широко используемый инструмент в экосистеме серверного рендеринга — Next.js. Это технология с открытым исходным кодом, выпущенная Zeit и призванная помочь инженерам упростить создание серверных приложений. Инструмент предоставляет функции для интуитивно понятной маршрутизации, статистической оптимизации, автоматического разделения и так далее.

В следующем разделе мы более подробно рассмотрим, как работать с Next.js, добавив в приложение серверный рендеринг.

Для начала создадим новый проект, выполнив следующие команды:

```
mkdir project-next
cd project-next
npm init -y
npm install --save react react-dom next
mkdir pages
```

Затем создадим несколько сценариев npm для более легкого выполнения общих команд:

```
{
  //...
  "scripts": {
    "dev": "next",
    "build": "next build",
    "start": "next start"
  }
}
```

В папке pages создадим файл index.js. Мы не будем импортировать React или ReactDOM. Вместо этого просто напишем компонент:

```
export default function Index() {
  return (
    <div>
      <p>Hello everyone!</p>
    </div>
  );
}
```

Затем запустим команду `npm run dev`, чтобы увидеть страницу, работающую на localhost:3000. Наш компонент отображается правильно.

В правом нижнем углу экрана есть небольшой значок молнии. При наведении курсора на него отобразится кнопка с надписью Prerendered Page. При щелчке по ней откроется документация Static Optimization Indicator. Это означает, что страница соответствует критериям автоматической статической оптимизации, и, соответственно, ее можно предварительно отобразить. Никакие требования к данным нам не мешают. Если страница автоматически статически оптимизирована (что сложно, но полезно!), она загружается быстрее, поскольку никаких усилий на стороне сервера не требуется. Страницу можно транслировать из CDN, что обеспечивает сверхбыстрое взаимодействие с пользователем. Не придется ничего делать, чтобы ощутить повышение производительности.

Но что, если у страницы есть требования к данным? Что делать, если страницу нельзя предварительно отобразить? Чтобы рассмотреть этот вопрос, сделаем приложение более надежным и построим компонент, который извлекает некоторые удаленные данные из API. Выполним это в новом файле Pets.js:

```
export default function Pets() {
  return <h1>Pets!</h1>;
}
```

Для начала отобразим `h1`. Теперь мы можем перейти по ссылке localhost:3000/pets и увидеть, что наша страница лежит именно там. Это хорошо, но можно сделать еще лучше, добавив ссылки и компонент макета, который будет отображать правильный контент для каждой страницы. Создадим заголовок, который будет использоваться на обеих страницах и отображать ссылки:

```
import Link from "next/link";

export default function Header() {
  return (
    <div>
      <Link href="/">
        <a>Home</a>
      </Link>
      <Link href="/pets">
        <a>Pets</a>
      </Link>
    </div>
  );
}
```

Компонент `Link` — это оболочка для пары ссылок. Они похожи на ссылки, которые мы создали с помощью React Router. Мы также можем добавить стиль к каждому из тегов `<a>`:

```
const linkStyle = {
  marginRight: 15,
  color: "salmon"
};

export default function Header() {
  return (
    <div>
      <Link href="/">
        <a style={linkStyle}>Home</a>
      </Link>
      <Link href="/pets">
        <a style={linkStyle}>Pets</a>
      </Link>
    </div>
  );
}
```

Далее включим компонент `Header` в новый файл Layout.js. Он будет динамически отображать компонент на основе правильного маршрута:

```
import Header from "./Header";

export function Layout(props) {
  return (
    <div>
      <Header />
      {props.children}
    </div>
  );
}
```

Компонент `Layout` будет принимать реквизиты и отображать контент в компоненте под `Header`. Затем на каждой странице мы создадим блоки контента, который можно передать компоненту `Layout` при рендеринге. Например, файл index.js теперь будет выглядеть так:

```
import Layout from "./Layout";

export default function Index() {
  return (
    <Layout>
      <div>
        <h1>Hello everyone!</h1>
      </div>
    </Layout>
  );
}
```

То же самое произойдет с файлом Pets.js:

```
import Layout from "./Layout";

export default function Pets() {
  return (
    <Layout>
      <div>
        <h1>Hey pets!</h1>
      </div>
    </Layout>
  );
}
```

Теперь, если мы зайдем на домашнюю страницу, то увидим заголовок, а нажав на ссылку Pets, увидим страницу с домашними животными.

Когда мы нажимаем кнопку с изображением молнии в правом нижнем углу, мы можем заметить, что эти страницы отображаются заранее. Этого и следовало ожидать, поскольку мы работаем со статическим контентом. Используем страницу Pets, чтобы загрузить некоторые данные и посмотреть, как они будут меняться.

Для начала установим библиотеку isomorphic-unetchfetch, как мы делали ранее в этой главе:

```
npm install isomorphic-unfetch
```

Мы будем использовать ее, чтобы вызвать Pet Library API. Начнем с импорта в файл Pages.js:

```
import fetch from "isomorphic-unfetch";
```

Затем добавим функцию `getInitialProps`. Она будет обрабатывать выборку и загрузку данных:

```
Pets.getInitialProps = async function() {
  const res = await fetch(
    `http://pet-library.moonhighway.com/api/pets`
  );
  const data = await res.json();
  return {
    pets: data
  };
};
```

Возвращая данные в качестве значения для `pets`, мы сможем сопоставить данные в компоненте.

Настроим компонент для сопоставления со свойством `pets`:

```
export default function Pets(props) {
  return (
    <Layout>
      <div>
        <h1>Pets!</h1>
        <ul>
          {props.pets.map(pet => (
            <li key={pet.id}>{pet.name}</li>
          ))}
        </ul>
      </div>
    </Layout>
  );
}
```

Если в компоненте есть функция `getInitialProps`, Next.js будет отображать страницу в ответ на каждый запрос. Это означает, что страница будет отображаться на стороне сервера вместо статического предварительного рендеринга, поэтому данные из API будут актуальными для каждого запроса.

Когда мы будем довольны готовностью приложения, мы сможем запустить сборку с помощью команды `npm run build`. Next.js заточен на производительность, поэтому он дает нам полную сводку количества килобайт для каждого файла. Это быстрая выборочная проверка файлов самого большого размера.

Рядом с каждым файлом мы увидим значок, указывающий, отображается ли сайт на сервере во время выполнения (λ), автоматически отображается как HTML (○) или автоматически создается как статический HTML + JSON (●).

Создав приложение, мы можем развернуть его. Next.js — это продукт Zeit с открытым исходным кодом, поэтому именно развертывание с помощью Zeit является наиболее простым. Однако для развертывания приложения можно использовать услуги разных хостинг-провайдеров.

Есть еще несколько терминов, которые важно понимать, если вы хотите создавать приложения:

CSR (рендеринг на стороне клиента)

Рендеринг приложения в браузере, как правило, с использованием DOM. Это то, что мы делаем по умолчанию в Create React App.

SSR (рендеринг на стороне сервера)

Рендеринг клиентского или универсального приложения в HTML на сервере.

Регидратация

Загрузка представлений JavaScript на клиенте для повторного использования обработанного сервером дерева DOM и данных HTML.

Предварительный рендеринг

Запуск клиентского приложения во время сборки и фиксация начального состояния в виде статического HTML.

Gatsby

Еще один популярный генератор сайтов, основанный на React, — это Gatsby. Gatsby набирает популярность как простой способ создания веб-сайта, ориентированного на контент. В нем есть разумные настройки по умолчанию для управления вопросами производительности, доступности, обработки изображений и многого другого. Если вы читаете эту книгу, вполне вероятно, что в какой-то момент вы сможете работать над проектом Gatsby!

Gatsby используется для ряда проектов, но чаще всего для создания веб-сайтов, ориентированных на контент. Другими словами, если у вас есть блог или статический контент, Gatsby вам подойдет, особенно теперь, когда вы знаете React. Gatsby также может обрабатывать динамический контент, например загрузку данных из API, интеграцию с фреймворками и многое другое.

В этом разделе мы начнем создавать сайт Gatsby, чтобы продемонстрировать, как он работает. По сути мы создадим наше приложение Next.js как приложение Gatsby:

```
npm install -g gatsby-cli
gatsby new pets
```

Если yarn установлен глобально, интерфейс командной строки спросит вас, что использовать: yarn или npm. Подойдет и то, и другое. Затем перейдите в папку pets:

```
cd pets
```

Теперь вы можете начать проект с помощью команды `gatsby develop`. Посетив адрес localhost:8000, вы увидите, что ваш стартовый сайт Gatsby запущен. Теперь вы можете просмотреть файлы.

Если вы откроете папку src проекта, то увидите три подпапки: components, images и pages.

В pages вы найдете страницы с ошибкой 404.js, index.js (страницу, которая отображается при посещении localhost:8000) и page-2.js, отображающую содержимое второй страницы.

В папке components скрыта вся магия Gatsby. Помните, как мы создавали компоненты `Header` и `Layout` с помощью Next.js? Оба эти компонента уже созданы как шаблоны в папке components.

Отметим несколько интересных моментов:

layout.js

: Содержит компонент `Layout`. В нем используется хук `useStaticQuery` для запроса некоторых данных о сайте с помощью GraphQL.

seo.js

: Позволяет получить доступ к метаданным страницы в целях поисковой оптимизации.

Если вы добавите в папку pages дополнительные страницы, они появятся на сайте. Добавим в папку pages файл page-3.js. Затем добавим в этот файл быстрый код для страницы:

```
import React from "react";
import { Link } from "gatsby";

import Layout from "../components/layout";
import SEO from "../components/seo";

const ThirdPage = () => (
  <Layout>
    <SEO title="Page three" />
```

```
    <h1>Hi from the third page</h1>
    <Link to="/">Go back to the homepage</Link>
  </Layout>
);

export default ThirdPage;
```

Мы используем компонент `Layout`, чтобы обернуть контент таким образом, чтобы он отображался как `children`. `Layout` не только отображает динамический контент, но и запускает автоматическое создание страницы.

Это лишь малая часть того, что вы можете делать с Gatsby, поэтому упомянем еще несколько его функций:

Статический контент

Позволяет создать свой сайт в виде статических файлов, которые можно развернуть без сервера.

Поддержка CDN

Позволяет кэшировать сайт в сетях CDN по всему миру, чтобы улучшить производительность и доступность.

Адаптивные и прогрессивные изображения

Загружает изображения как размытые заполнители, а затем постепенно загружет четкую картинку. Эта тактика, популяризированная Medium, позволяет пользователям увидеть рендеринг чего-либо до того, как будет доступен полный ресурс.

Предварительная загрузка связанных страниц

Загружает в фоновом режиме весь контент, необходимый для загрузки следующей страницы, прежде чем вы нажмете на следующую ссылку.

Все эти и многие другие функции используются для обеспечения беспрепятственного взаимодействия с пользователем. Gatsby принимает множество решений за вас. Это может быть хорошо или плохо, но зато эти ограничения позволят вам сосредоточиться на своем контенте.

Будущее React

Хотя системы Angular, Ember и Vue по-прежнему занимают существенную долю рынка в экосистеме JavaScript, трудно спорить с тем фактом, что React сегодня является наиболее широко используемой и влиятельной библиотекой

для создания приложений JavaScript. Популярность библиотеки поддерживает огромное сообщество JavaScript, о чем свидетельствует, в частности, появление Next.js и Gatsby.

Что дальше? Мы рекомендуем вам использовать полученные при чтении этой книги навыки для создания собственных проектов. Если вы хотите создавать мобильные приложения, вы можете попробовать React Native. Если хотите получать данные декларативно, попробуйте GraphQL. Чтобы создавать веб-сайты на основе контента, углубитесь в Next.js и Gatsby.

Есть несколько путей, по которым вы можете пройти, но полученные навыки работы с React точно вам пригодятся, когда вы начнете создавать свои приложения. И мы надеемся, что эта книга послужит справочником. Хотя React и связанные с ней библиотеки почти наверняка изменятся, их изменения обычно можно сразу использовать. Создание приложений с помощью React и функционального декларативного JavaScript — это очень весело, и нам не терпится увидеть, что вы создадите.

Об авторах

Алекс Бэнкс и Ева Порселло — разработчики, программисты, преподаватели и соучредители Moon Highway, компании по разработке учебных программ в Северной Калифорнии. Они являются авторами курсов для LinkedIn Learning и egghead.io, часто выступают на конференциях и проводят семинары для специалистов всего мира.

Об обложке

Животное на обложке книги — кабан с детенышами (sus scrofa). Кабан, так же известный как дикая свинья, или евразийская дикая свинья, обитает в Евразии, Северной Африке и на Больших Зондских островах. Благодаря вмешательству человека они стали одним из самых многочисленных видов млекопитающих в мире.

У кабанов короткие тонкие ноги и массивное туловище. Короткая толстая шея и большая голова, составляющая до трети длины тела. Размеры и вес взрослых особей зависят от факторов окружающей среды, таких как доступ к пище и воде. Несмотря на свой размер, кабаны развивают скорость до 40 км в час и прыгают на высоту 1,4–1,5 м. Зимой их шерсть состоит из грубой щетины, под которой спрятан короткий коричневый пушистый мех. Самые длинные щетинки находятся на спине кабана, а самые короткие — на морде и конечностях.

У кабанов сильно развито обоняние, и в Германии их использовали для обнаружения наркотических веществ. У них острый слух, который компенсирует слабое зрение и неспособность различать цвета: они не могут распознать человека, стоящего на расстоянии 9 м от них.

Кабаны — социальные животные, живущие группами, в которых доминируют самки. Период активного размножения кабанов длится примерно с ноября по январь. При подготовке к спариванию самцы претерпевают несколько телесных изменений, включая развитие подкожного панциря, который помогает во время столкновений с соперниками. В поисках самки кабаны путешествуют на большие расстояния и по пути едят очень мало. В среднем помете содержится от четырех до шести поросят.

Многие животные на обложках O'Reilly находятся под угрозой исчезновения, все они важны для мира.

Иллюстрация на обложке сделана Карен Монтгомери на основе черно-белой гравюры из Meyers Kleines Lexicon.

Алекс Бэнкс, Ева Порселло

React: современные шаблоны для разработки приложений

2-е издание

Перевел с английского *С. Черников*

Заведующая редакцией	*Ю. Сергиенко*
Ведущий редактор	*А. Юринова*
Литературный редактор	*А. Руденко*
Художественный редактор	*В. Мостипан*
Корректоры	*С. Беляева, Н. Сидорова*
Верстка	*Л. Егорова*

Изготовлено в России. Изготовитель: ООО «Прогресс книга».
Место нахождения и фактический адрес: 194044, Россия, г. Санкт-Петербург,
Б. Сампсониевский пр., д. 29А, пом. 52. Тел.: +78127037373.

Дата изготовления: 08.2021. Наименование: книжная продукция. Срок годности: не ограничен.

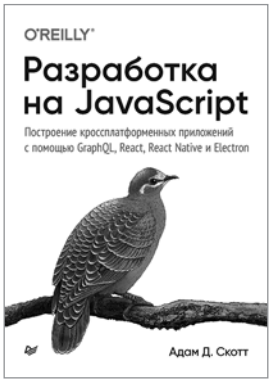
Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12 —
Книги печатные профессиональные, технические и научные.

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 23.07.21. Формат 70×100/16. Бумага офсетная. Усл. п. л. 25,800. Тираж 700. Заказ 0000.

Адам Д. Скотт

РАЗРАБОТКА НА JAVASCRIPT. ПОСТРОЕНИЕ КРОССПЛАТФОРМЕННЫХ ПРИЛОЖЕНИЙ С ПОМОЩЬЮ GRAPHQL, REACT, REACT NATIVE И ELECTRON



Что такое современный JavaScript? Когда-то он просто добавлял интерактивности к окнам веб-браузера, а теперь превратился в основательный фундамент мощного и надежного софта. Разработчики любого уровня смогут использовать JavaScript для создания API, веб-, мобильных и десктопных приложений.

В этой книге:

- Работа с данными с помощью GraphQL.
- Аутентификация для API, веб- и нативных приложений.
- Создание высокопроизводительных веб-приложений.
- Разработка кроссплатформенных приложений под iOS и Android.
- Создание десктопных приложений.

Борис Черный

ПРОФЕССИОНАЛЬНЫЙ TYPESCRIPT. РАЗРАБОТКА МАСШТАБИРУЕМЫХ JAVASCRIPT-ПРИЛОЖЕНИЙ

Любой программист, работающий с языком с динамической типизацией, подтвердит, что задача масштабирования кода невероятно сложна и требует большой команды инженеров. Вот почему Facebook, Google и Microsoft придумали статическую типизацию для динамически типизированного кода.

Работая с любым языком программирования, мы отслеживаем исключения и вычитываем код строку за строкой в поиске неисправности и способа ее устранения. TypeScript позволяет автоматизировать эту неприятную часть процесса разработки.

TypeScript, в отличие от множества других типизированных языков, ориентирован на прикладные задачи. Он вводит новые концепции, позволяющие выражать идеи более кратко и точно и легко создавать масштабируемые и безопасные современные приложения.

Борис Черный помогает разобраться со всеми нюансами и возможностями TypeScript, учит устранять ошибки и масштабировать код.