React
Сборник рецептов

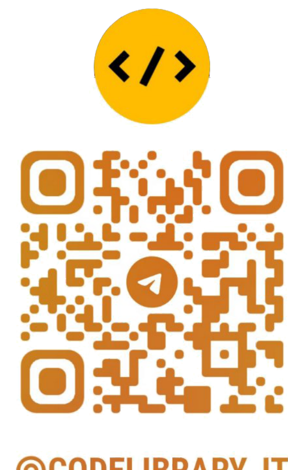
Практические приемы работы с фреймворком React

Дэвид Гриффитс, Дон Гриффитс

React Cookbook

Recipes for Mastering the React Framework

David Griffiths and Dawn Griffiths



Beijing • Boston • Farnham • Sebastopol • Tokyo

Дэвид Гриффитс
Дон Гриффитс

React
Сборник рецептов

Санкт-Петербург
«БХВ-Петербург»
2023

УДК 004.438
ББК 32.973.26-018.1
Г85

Гриффитс, Д.

Г85 React. Сборник рецептов: Пер. с англ. / Дэвид Гриффитс, Дон Гриффитс. — СПб.: БХВ-Петербург, 2023. — 528 с.: ил.

ISBN 978-5-9775-6839-5

Книга посвящена практическому применению фреймворка React. Описано создание простых приложений и приложений со сложным интерфейсом, рассмотрены вопросы маршрутизации в приложениях и управление их состоянием. Даны примеры реализации интерактивного взаимодействия с пользователем, подключения к различным службам бэкенда, таким как REST и GraphQL, описана работа с библиотеками компонентов. Подробно рассматривается безопасность приложений, процесс их тестирования, даны советы по обеспечению доступности. Приводятся практические рекомендации по повышению производительности и созданию прогрессивных веб-приложений.

Для программистов

УДК 004.438
ББК 32.973.26-018.1

Группа подготовки издания:

Руководитель проекта *Павел Шалин*
Зав редакцией *Людмила Гауль*
Компьютерная верстка *Ольги Сергиенко*
Оформление обложки *Зои Канторович*

© 2022 BHV
Authorized Russian translation of the English edition of ***React Cookbook*** ISBN 9781492085843
© 2021 Dawn Griffiths and David Griffiths
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.

Авторизованный перевод с английского языка на русский издания ***React Cookbook*** ISBN 9781492085843
© 2021 Dawn Griffiths and David Griffiths.
Перевод опубликован и продается с разрешения компании-правообладателя O'Reilly Media, Inc

Подписано в печать 03 10 22
Формат 70×100 1/16 Печать офсетная Усл печ л 42,57
Тираж 1300 экз Заказ № 5394.
"БХВ-Петербург", 191036, Санкт-Петербург, Гончарная ул , 20

Отпечатано с готового оригинал-макета
ООО "Принт-М", 142300, М О , г. Чехов, ул Полиграфистов, д 1

ISBN 978-1-492-08584-3 (англ.)
ISBN 978-5-9775-6839-5 (рус.)

© Dawn Griffiths, David Griffiths, 2021
© Перевод на русский язык, оформление ООО "БХВ-Петербург", ООО "БХВ", 2023

Оглавление

Введение 13
Типографские соглашения 15
Использование примеров кода 15
Возможности онлайнового обучения от компании O'Reilly 16
Как связаться с нами 16
Благодарности 17

Глава 1. Создание приложений 19
1.1. Создаем простое приложение 19
ЗАДАЧА 19
РЕШЕНИЕ 19
Обсуждение 23
1.2. Создание приложений с обширным информационным наполнением посредством Gatsby 24
ЗАДАЧА 24
РЕШЕНИЕ 24
Обсуждение 27
1.3. Создание универсальных приложений с помощью Razzle 28
ЗАДАЧА 28
РЕШЕНИЕ 28
Обсуждение 30
1.4. Создание серверного и клиентского кода посредством Next.js 30
ЗАДАЧА 30
РЕШЕНИЕ 31
Обсуждение 32
1.5. Создание крошечных приложений посредством Preact 33
ЗАДАЧА 33
РЕШЕНИЕ 33
Обсуждение 36
1.6. Создание библиотек посредством набора инструментов *nwb* 37
ЗАДАЧА 37
РЕШЕНИЕ 37
Обсуждение 39
1.7. Добавление React в код Rails посредством Webpacker 39
ЗАДАЧА 39
РЕШЕНИЕ 39
Обсуждение 41
1.8. Создание пользовательских элементов посредством Preact 41
ЗАДАЧА 41
РЕШЕНИЕ 41
Обсуждение 44

1.9. Разработка компонентов посредством Storybook 45
ЗАДАЧА 45
РЕШЕНИЕ 45
Обсуждение 48
1.10. Тестирование кода в браузере посредством Cypress 48
ЗАДАЧА 48
РЕШЕНИЕ 48
Обсуждение 49

Глава 2. Маршрутизация 51
2.1. Создание интерфейсов, используя реагирующую маршрутизацию 51
ЗАДАЧА 51
РЕШЕНИЕ 52
Обсуждение 58
2.2. Размещение состояния в маршрутах 59
ЗАДАЧА 59
РЕШЕНИЕ 60
Обсуждение 64
2.3. Модульное тестирование посредством MemoryRouter 65
ЗАДАЧА 65
РЕШЕНИЕ 66
Обсуждение 67
2.4. Подтверждение ухода со страницы посредством компонента Prompt 68
ЗАДАЧА 68
РЕШЕНИЕ 69
Обсуждение 75
2.5. Создание переходов посредством библиотеки React Transition Group 75
ЗАДАЧА 75
РЕШЕНИЕ 76
Обсуждение 81
2.6. Создание защищенных маршрутов 81
ЗАДАЧА 81
РЕШЕНИЕ 82
Обсуждение 86

Глава 3. Управление состоянием 87
3.1. Управление сложным состоянием посредством преобразователей 87
ЗАДАЧА 87
РЕШЕНИЕ 88
Обсуждение 94
3.2. Создание возможности "Отмена" 95
ЗАДАЧА 95
РЕШЕНИЕ 95
Обсуждение 101
3.3. Создание форм и проверка действительности их данных 102
ЗАДАЧА 102
РЕШЕНИЕ 102
Обсуждение 110
3.4. Часы для измерения времени 110
ЗАДАЧА 110
РЕШЕНИЕ 110
Обсуждение 113

3.5. Мониторинг состояния сетевого подключения 114
ЗАДАЧА 114
РЕШЕНИЕ 114
Обсуждение 116
3.6. Управление глобальным состоянием посредством библиотеки Redux 116
ЗАДАЧА 116
РЕШЕНИЕ 116
Обсуждение 122
3.7. Сохранение состояния при обновлении страниц посредством Redux Persist 123
ЗАДАЧА 123
РЕШЕНИЕ 124
Обсуждение 127
3.8. Вычисление производного состояния посредством Reselect 127
ЗАДАЧА 127
РЕШЕНИЕ 128
Обсуждение 131

Глава 4. Проектирование для обеспечения интерактивности 133
4.1. Создание центрального обработчика ошибок 133
ЗАДАЧА 133
РЕШЕНИЕ 134
Обсуждение 138
4.2. Создаем интерактивное справочное руководство 139
ЗАДАЧА 139
РЕШЕНИЕ 139
Обсуждение 146
4.3. Сложные взаимодействия посредством преобразователей 146
ЗАДАЧА 146
РЕШЕНИЕ 147
Обсуждение 153
4.4. Взаимодействие с клавиатурой 153
ЗАДАЧА 153
РЕШЕНИЕ 153
Обсуждение 156
4.5. Создание насыщенного содержимого посредством редактора Markdown 156
ЗАДАЧА 156
РЕШЕНИЕ 156
Обсуждение 161
4.6. Анимация посредством классов CSS 161
ЗАДАЧА 161
РЕШЕНИЕ 161
Обсуждение 163
4.7. Анимация средствами React 163
ЗАДАЧА 163
РЕШЕНИЕ 164
Обсуждение 168
4.8. Анимация информационной графики посредством библиотеки TweenOne 168
ЗАДАЧА 168
РЕШЕНИЕ 169
Обсуждение 175

Глава 5. Подключение к службам ... 176
5.1. Преобразование сетевых вызовов в хуки ... 176
ЗАДАЧА ... 176
РЕШЕНИЕ ... 177
Обсуждение ... 182
5.2. Автоматическое обновление посредством счетчиков состояния ... 183
ЗАДАЧА ... 183
РЕШЕНИЕ ... 184
Обсуждение ... 191
5.3. Отмена сетевых запросов посредством маркеров ... 192
ЗАДАЧА ... 192
РЕШЕНИЕ ... 192
Обсуждение ... 195
5.4. Сетевые вызовы посредством Redux ... 195
ЗАДАЧА ... 195
РЕШЕНИЕ ... 196
Обсуждение ... 201
5.5. Подключение к GraphQL ... 202
ЗАДАЧА ... 202
РЕШЕНИЕ ... 202
Обсуждение ... 209
5.6. Уменьшение сетевой нагрузки при помощи очищенных запросов ... 210
ЗАДАЧА ... 210
РЕШЕНИЕ ... 211
Обсуждение ... 213

Глава 6. Библиотеки компонентов ... 214
6.1. Использование библиотеки Material Design совместно с библиотекой Material-UI ... 215
ЗАДАЧА ... 215
РЕШЕНИЕ ... 215
Обсуждение ... 222
6.2. Простой пользовательский интерфейс посредством React Bootstrap ... 223
ЗАДАЧА ... 223
РЕШЕНИЕ ... 223
Обсуждение ... 227
6.3. Просмотр наборов данных посредством окна React Window ... 227
ЗАДАЧА ... 227
РЕШЕНИЕ ... 227
Обсуждение ... 230
6.4. Создание реагирующих диалоговых окон посредством библиотеки Material-UI ... 230
ЗАДАЧА ... 230
РЕШЕНИЕ ... 230
Обсуждение ... 232
6.5. Создание консоли администратора посредством React Admin ... 233
ЗАДАЧА ... 233
РЕШЕНИЕ ... 233
Обсуждение ... 240
6.6. Использование Semantic UI вместо дизайнера ... 240
ЗАДАЧА ... 240
РЕШЕНИЕ ... 241
Обсуждение ... 246

Глава 7. Безопасность ... 247
7.1. Защищаем запросы, а не маршруты ... 247
ЗАДАЧА ... 247
РЕШЕНИЕ ... 247
Обсуждение ... 255
7.2. Аутентификация посредством физических ключей ... 256
ЗАДАЧА ... 256
РЕШЕНИЕ ... 256
Обсуждение ... 266
7.3. Работа с протоколом HTTPS ... 267
ЗАДАЧА ... 267
РЕШЕНИЕ ... 267
Обсуждение ... 271
7.4. Аутентификация посредством отпечатка пальца ... 271
ЗАДАЧА ... 271
РЕШЕНИЕ ... 271
Обсуждение ... 276
7.5. Подтверждение действий, предоставляя учетные данные ... 277
ЗАДАЧА ... 277
РЕШЕНИЕ ... 277
Обсуждение ... 283
7.6. Однофакторная аутентификация ... 284
ЗАДАЧА ... 284
РЕШЕНИЕ ... 284
Обсуждение ... 288
7.7. Проверка приложения на устройстве Android ... 289
ЗАДАЧА ... 289
РЕШЕНИЕ ... 290
Обсуждение ... 291
7.8. Проверка безопасности посредством ESlint ... 291
ЗАДАЧА ... 291
РЕШЕНИЕ ... 292
Обсуждение ... 295
7.9. Удобные для браузера формы входа в систему ... 296
ЗАДАЧА ... 296
РЕШЕНИЕ ... 296
Обсуждение ... 297

Глава 8. Тестирование ... 299
8.1. Работа с библиотекой React Testing Library ... 300
ЗАДАЧА ... 300
РЕШЕНИЕ ... 301
Обсуждение ... 307
8.2. Использование Storybook для тестирования отрисовки ... 308
ЗАДАЧА ... 308
РЕШЕНИЕ ... 308
Обсуждение ... 314
8.3. Тестирование без сервера посредством Cypress ... 315
ЗАДАЧА ... 315
РЕШЕНИЕ ... 315
Обсуждение ... 322

8.4. Использование Cypress для офлайнового тестирования 323
ЗАДАЧА 323
РЕШЕНИЕ 323
Обсуждение 326
8.5. Использование Selenium для тестирования в браузере 326
ЗАДАЧА 326
РЕШЕНИЕ 327
Обсуждение 333
8.6. Тестирование внешнего вида на разных браузерах посредством ImageMagick 334
ЗАДАЧА 334
РЕШЕНИЕ 334
Обсуждение 341
8.7. Добавление консоли в браузер мобильного устройства 342
ЗАДАЧА 342
РЕШЕНИЕ 342
Обсуждение 345
8.8. Удаление произвольности из тестов 346
ЗАДАЧА 346
РЕШЕНИЕ 347
Обсуждение 349
8.9. Путешествие во времени 350
ЗАДАЧА 350
РЕШЕНИЕ 350
Обсуждение 355

Глава 9. Доступность специальных возможностей 356
9.1. Использование ориентиров 357
ЗАДАЧА 357
РЕШЕНИЕ 359
Обсуждение 362
9.2. Применение ролей 362
ЗАДАЧА 362
РЕШЕНИЕ 363
Обсуждение 371
9.3. Проверка доступности посредством ESlint 372
ЗАДАЧА 372
РЕШЕНИЕ 372
Обсуждение 378
9.4. Динамический анализ посредством axe DevTools 379
ЗАДАЧА 379
РЕШЕНИЕ 379
Обсуждение 383
9.5. Автоматизация тестирования в браузере посредством Cypress Axe 384
ЗАДАЧА 384
РЕШЕНИЕ 384
Обсуждение 387
9.6. Добавление в страницу кнопок пропуска содержимого 388
ЗАДАЧА 388
РЕШЕНИЕ 389
Обсуждение 394

9.7. Добавление возможности пропуска областей страницы 395
ЗАДАЧА 395
РЕШЕНИЕ 396
Обсуждение 404
9.8. Захват области действия в модальных окнах 404
ЗАДАЧА 404
РЕШЕНИЕ 406
Обсуждение 407
9.9. Создание считывателя экрана посредством Speech API 407
ЗАДАЧА 407
РЕШЕНИЕ 408
Обсуждение 412

Глава 10. Производительность 413
10.1. Браузерные средства настройки производительности 414
ЗАДАЧА 414
РЕШЕНИЕ 414
Обсуждение 421
10.2. Слежение за отрисовкой посредством *Profiler* 421
ЗАДАЧА 421
РЕШЕНИЕ 422
Обсуждение 427
10.3. Создание модульных тестов с *Profiler* 427
ЗАДАЧА 427
РЕШЕНИЕ 427
Обсуждение 431
10.4. Точное измерение времени 432
ЗАДАЧА 432
РЕШЕНИЕ 433
Обсуждение 434
10.5. Уменьшение размера приложений посредством разделения кода 436
ЗАДАЧА 436
РЕШЕНИЕ 437
Обсуждение 442
10.6. Объединение сетевых обещаний 443
ЗАДАЧА 443
РЕШЕНИЕ 444
Обсуждение 446
10.7. Отрисовка на стороне сервера 447
ЗАДАЧА 447
РЕШЕНИЕ 447
Обсуждение 457
10.8. Использование основных показателей веб-производительности 458
ЗАДАЧА 458
РЕШЕНИЕ 459
Обсуждение 461

Глава 11. Прогрессивные веб-приложения 462
11.1. Создаем сервис-воркеры посредством Workbox 462
ЗАДАЧА 462
РЕШЕНИЕ 465
Обсуждение 479

11.2. Создание прогрессивных веб-приложений посредством Create React App....480
ЗАДАЧА 480
РЕШЕНИЕ....480
Обсуждение 483
11.3. Кеширование сторонних ресурсов....483
ЗАДАЧА 483
РЕШЕНИЕ....483
Обсуждение 487
11.4. Автоматическая перезагрузка воркеров 487
ЗАДАЧА 487
РЕШЕНИЕ....489
Обсуждение 492
11.5. Добавление извещений 492
ЗАДАЧА 492
РЕШЕНИЕ....493
Обсуждение 499
11.6. Модификации в режиме офлайн посредством фоновой синхронизации 500
ЗАДАЧА 500
РЕШЕНИЕ....500
Обсуждение 505
11.7. Добавляем специализированный установочный пользовательский интерфейс 506
ЗАДАЧА 506
РЕШЕНИЕ....507
Обсуждение 510
11.8. Предоставление ответов в режиме офлайн 511
ЗАДАЧА 511
РЕШЕНИЕ....512
Обсуждение 515

Предметный указатель....517

Об авторах....525

Об обложке....526

Введение

Эта книга содержит коллекцию фрагментов кода, которые в результате многолетнего опыта мы считаем полезными для разработки приложений React. Подобно кулинарным рецептам, эти примеры предназначены для использования в качестве отправной точки при создании собственных программ. Их следует модифицировать под конкретную ситуацию и заменять любые составляющие части (например, сервер примеров) на те, которые лучше отвечают вашим требованиям. Приведен обширный ассортимент рецептов — от общих советов по разработке веб-приложений до фрагментов кода более большого размера, которые можно обобщить в библиотеки.

Большинство примеров созданы с помощью приложения Create React App. В настоящее время это распространенный подход при разработке проектов React. Каждый рецепт можно легко преобразовать для Preact или Gatsby.

Для сокращения объема кода мы обычно использовали хуки (hooks) и функции, а не классовые компоненты. Мы также применяли средство Prettier с настройками по умолчанию, чтобы наложить стандартное форматирование на весь код. Единственное изменение состоит в уменьшении отступов и укорочении строк, чтобы поместить листинги на книжную страницу. Вы можете настроить формат кода в соответствии со своими предпочтениями.

В табл. 1 приведены необходимые инструменты и библиотеки.

Таблица 1

Средство или библиотека	Описание	Версии
Apollo Client	Клиент GraphQL	3.3.19
axios	Библиотека HTTP	0.21.1
chai	Вспомогательная библиотека для проверки модулей	4.3.0
chromedriver	Инструмент для автоматизации проверки работы приложений с разными браузерами	88.0.0
Create React App	Инструмент для создания приложений React	4.0.3
Cypress	Автоматизированная система тестирования	7.3.0
Cypress Axe	Авторизированная система тестирования доступности	0.12.2
Gatsby	Инструмент для создания приложений React	3.4.1

Таблица 1 (продолжение)

Средство или библиотека	Описание	Версии
GraphQL	Язык запросов для API-интерфейсов	15.5.0
jsx-a11y	Подключаемый модуль ESLint для обеспечения доступности	6.4.1
Material-UI	Библиотека компонентов	4.11.4
Node	Среда исполнения для JavaScript	v12.20.0
npm	Диспетчер пакетов для Node	6.14.8
nvm	Инструмент для исполнения нескольких сред Node	0.33.2
nwb	Инструмент для создания приложений React	0.25 *x*
Next.js	Инструмент для создания приложений React	10.2.0
Preact	Облегченная версия фреймворка наподобие React	10.3.2
Preact Custom Elements	Библиотека для создания пользовательских элементов	4.2.1
preset-create-react-app	Подключаемый модуль для Storybook	3.1.7
Rails	Фреймворк для разработки веб-приложений	6.0.3.7
Razzle	Инструмент для создания приложений React	4 0.4
React	Фреймворк для разработки веб-приложений	17.0.2
React Media	Компонент CSS для медиавыражений в коде React	1.10.0
React Router (DOM)	Библиотека для управления маршрутами React	5.2.0
React Testing Library	Библиотека для тестирования модулей React	11.1.0
react-animations	Библиотека CSS-анимации для React	1 0.0
React Focus Lock	Библиотека для захвата фокуса клавиатуры	2.5.0
react-md-editor	Редактор облегченного языка разметки Markdown	3.3.6
React-Redux	Библиотека поддержки React для Redux	7.2.2
Redux	Библиотека управления состояниями	4.0.5
Redux-Persist	Библиотека для хранения состояния Redux	6.0.0
Ruby	Язык, используемый во фреймворке Rails	2.7.0p0
selenium-webdriver	Фреймворк для тестирования веб-приложений в разных браузерах	4.0.0-beta.1
Storybook	Система галереи компонентов	6.2.9
TweenOne	Библиотека анимации для React	2.7.3
Typescript	Расширения для обеспечения безопасности типов для JavaScript	4.1 2
Webpacker	Инструмент для добавления React в приложения Rails	4.3.0
Workbox	Библиотека для создания сервис-воркеров	5.1.3
Yarn	Диспетчер пакетов для Node	1.22.10

Типографские соглашения

В книге используются следующие типографские соглашения:

- *Курсив* — для обозначения новых терминов, адресов URL и электронной почты, названий и расширений файлов.
- `Моноширинный шрифт` — для листингов программ, а также в тексте для обозначения элементов программ, например названий функций, баз данных, типов данных, переменных среды, операторов и ключевых слов.
- **`Моноширинный полужирный шрифт`** — для обозначения команд или другого текста, который вводится пользователем.
- *`Моноширинный курсив`* — для обозначения текста, который нужно заменить предоставляемыми пользователем значениями или значениями, определяемыми контекстом.

Данное изображение обозначает подсказку или совет.

Данное изображение обозначает общее замечание.

Данное изображение обозначает предупреждение или предостережение.

Использование примеров кода

Дополнительный материал для книги (примеры кода, упражнения и т. п.) можно загрузить по адресу **https://github.com/dogriffiths/ReactCookbook-source**.

Вопросы по техническим аспектам примеров кода или проблемам с их использованием можно задать, отправив сообщение электронной почтой по адресу **bookquestion@oreily.com**.

Эта книга предназначена для того, чтобы помочь вам реализовать свои проекты. В общем, предоставляемые в этой книге примеры кода можно использовать в своих программах и документации, не спрашивая нашего разрешения, если только речь не идет о значительном объеме кода. Например, для использования нескольких фрагментов кода из этой книги в своей программе разрешения не требуется, а вот для продажи или распространения примеров из книг издательства O'Reilly такое разрешение необходимо. Цитирование теста из этой книги, включая код, разрешения не требует, но для включения значительного объема кода из книги в документацию своего продукта требуется разрешение.

При цитировании материалов из наших книг мы будем признательны за предоставление ссылки на источник, но обычно не требуем этого. Формат такой ссылки может включать название, имя автора, издателя и ISBN книги. Например: "React Cookbook by David Griffiths and Dawn Griffiths (O'Reilly). Copyright 2021 Dawn Griffiths and David Griffiths, 978-1-492-08584-3".

Если вы подозреваете, что ваше заимствование примеров кода может выходить за рамки принципа добросовестного использования или данных выше разрешений, то можете уточнить этот вопрос электронной почтой по адресу **permissions@oreilly.com**.

Возможности онлайнового обучения от компании O'Reilly

В течение свыше 40 лет компания O'Reilly Media предоставляет возможности обучения в области технологий и бизнеса, распространяя знания и сведения, помогающие компаниям добиваться успеха.

Мы создали уникальное сообщество экспертных специалистов и новаторов, которые делятся своими знаниями и опытом посредством книг, статей и нашей онлайновой обучающей платформы. Онлайновая система обучения компании O'Reilly предоставляет доступ по запросу к учебным курсам в режиме реального времени, разнообразным обучающим возможностям, интерактивным средам кодирования, а также к огромному собранию печатного и видеоматериала от самой компании и от свыше 200 других издательств. Чтобы получить дополнительную информацию, посетите наш веб-сайт по адресу **http://oreilly.com**.

Как связаться с нами

Если у вас есть какие-либо замечания или вопросы по этой книге, вы может задать их издателю по следующему адресу:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (в США или Канаде)
707-829-0515 (международный или местный)
707-829-0104 (факс)

Для этой книги создан веб-сайт, который содержит список ошибок, там же приводятся примеры и предоставляется другая дополнительная информация. Веб-сайт находится по адресу **https://oreil.ly/react-cb**.

Свои комментарии о книге и технические вопросы присылайте электронной почтой по адресу **bookquestions@oreilly.com**.

Чтобы узнать последние новости и информацию о наших книгах и курсах, посетите наш веб-сайт по адресу **http://oreilly.com**.

Мы на Facebook: **http://facebook.com/oreilly**.

И в Twitter: **http://twitter.com/oreillymedia**.

А также на YouTube: **http://www.youtube.com/oreillymedia**.

Благодарности

Мы хотим поблагодарить нашего очень терпеливого редактора Корбина Коллинса (Corbin Collins) за его помощь и советы в течение последнего года. Его спокойствие и доброжелательность были стабилизирующими факторами в процессе подготовки материала этой книги. Мы также хотим выразить свою благодарность главному редактору компании O'Reilly Аманде Куинн (Amanda Quinn) за приобретение нашей книги, а также Дэнни Эльфанбауму (Danny Elfanbaum) и производственной бригаде компании O'Reilly за воплощение в реальность физической и электронной версий книги.

Особое спасибо хотим сказать Сэму Уорнеру (Sam Warner) и Марку Хобсону (Mark Hobson) за их скрупулезную проверку материала в данной книге.

Мы также благодарны разработчикам многих библиотек с открытым кодом, поддерживающих экосистему React. Огромное спасибо им всем, особенно за их оперативное реагирование на сообщения об ошибках или просьбы о помощи.

Если наши рецепты окажутся полезными для вас, то это в первую очередь благодаря работе этих людей. Но ответственность за все ошибки в коде и тексте мы берем полностью на себя.

ГЛАВА 1

Создание приложений

Фреймворк React — на удивление гибкая платформа разработки. Разработчики могут использовать ее для создания как одностраничных приложений (Single-Page Application — SPA), с большим содержанием кода JavaScript, так и поразительно небольших подключаемых модулей. С ее помощью можно также вставлять код в приложения Rails или создавать веб-сайты с богатым информационным наполнением.

В этой главе мы рассмотрим разные способы создания приложений React, а также некоторые полезные инструменты, которые пригодятся вам в процессе разработки. Очень немногие разработчики начинают свои проекты JavaScript с чистого листа, т. к. это длительный и трудоемкий процесс, требующий большого объема настроек и "танцев с бубном". К счастью, почти в каждом случае требуемый программный код можно создать с помощью специального инструмента.

Вкратце рассмотрим эти многочисленные способы создания начального приложения React, начиная с наиболее часто применяемого средства: `create-react-app`.

1.1. Создаем простое приложение

ЗАДАЧА

Приложения React трудно создавать и настраивать с чистого листа. Это обусловлено необходимостью принятия многих предварительных решений: какие использовать библиотеки, средства, возможности языка и т. п. Кроме того, природа созданных вручную приложений такова, что они отличаются одно от другого. Специфические особенности конкретных проектов увеличивают время, необходимое для того, чтобы новый разработчик вышел на эффективный уровень работы.

РЕШЕНИЕ

Средство `create-react-app` предназначено для создания одностраничных приложений с типовой структурой и оптимальным набором стандартных опций. Для создания, тестирования и исполнения кода созданных таким образом проектов предназначена библиотека React Scripts. В проектах задаются стандартные настройки сборщика пакетов Webpack и подключается стандартный набор языковых возможностей.

Всякий разработчик, которому приходилось сталкиваться с одним приложением, созданным посредством инструмента create-react-app, сразу же может начать работу с любым другим таким приложением без каких бы то ни было проблем. Он понимает структуру проекта и знает, какие возможности языка можно использовать. Приложение содержит все возможности, присущие типичному приложению: от конфигурации компилятора Babel и загрузчиков файлов до библиотек для тестирования и сервера для разработки.

Если вам никогда раньше не приходилось работать с React или нужно создать типичное одностраничное приложение с минимальными усилиями, тогда мы советуем вам рассмотреть применение средства create-react-app для разработки своего приложения.

Один из вариантов использования средства create-react-app — установить его глобально на своем компьютере, но в настоящее время такой подход не рекомендуется. Вместо этого следует создавать новые проекты, вызывая create-react-app через утилиту npx для запуска npm пакетов. Такой подход обеспечивает создание приложения с помощью самой последней версии средства create-react-app:

```
$ npx create-react-app my-app
```

Исполнение этой команды создает папку с новым проектом, называющуюся my-app. По умолчанию в приложении задан JavaScript. Если требуется поменять язык на TypeScript, то вызов npx осуществляется следующим образом:

```
$ npx create-react-app --template typescript my-app
```

Средство create-react-app было разработано компанией Facebook, поэтому не удивительно, что если у вас установлен диспетчер пакетов yarn, то в новом проекте он будет назначен по умолчанию. Чтобы задействовать диспетчер пакетов npm, можно или указать флаг --use-npm, или же перейти в папку проекта, удалить файл yarn.lock, а затем снова выполнить установку, используя npm:

```
$ cd my-app
$ rm yarn.lock
$ npm install
```

Для запуска созданного приложения нужно выполнить следующую команду:

```
$ npm start # или yarn start
```

В результате запускается веб-сервер на порту 3000 и в браузере открывается домашняя страница, как показано на рис. 1.1.

Сервер предоставляет приложение в виде одного большого пакета кода JavaScript. Код устанавливает все свои компоненты в этом теге <div/> в файле public/index.html:

```
<div id="root"></div>
```

Начальный код для генерирования компонентов приложения находится в файле src/index.js (или в файле src/index.tsx, если используется TypeScript). Этот код приведен в листинге 1.1.

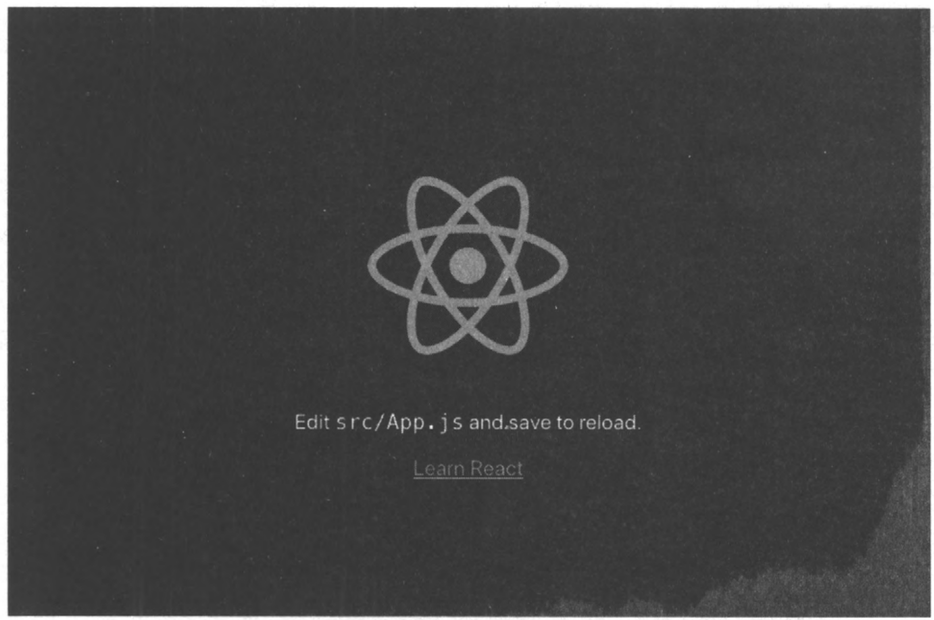


Рис. 1.1. Домашняя страница нового приложения

Листинг 1.1. Начальный код для генерирования компонентов приложения

```
import React from 'react'
import ReactDOM from 'react-dom'
import './index.css'
import App from './App'
import reportWebVitals from './reportWebVitals'

ReactDOM.render(
  <React.StrictMode>
    <App />
  </React.StrictMode>,
  document.getElementById('root')
)

// Если вы хотите выполнять замеры поддерживаемых метрик
// производительности в своем приложении, передайте соответствующую
// функцию в функцию reportWebVitals(), чтобы сохранить результаты
// в журнале (например, reportWebVitals(console.log)),
// или же отправьте ее на узел аналитики (Google).
// Дополнительную информацию см. по адресу: https://bit.ly/CRA-vitals
reportWebVitals()
```

Код из листинга 1.1 всего лишь прорисовывает один компонент, называющийся `<App/>`, который он импортирует из файла App.js (или App.tsx), находящегося в этой же папке. Код этого компонента приведен в листинге 1.2.

Листинг 1.2. Код компонента <App/>

```
import logo from './logo.svg'
import './App.css'

function App() {
  return (
    <div className="App">
      <header className="App-header">
        <img src={logo} className="App-logo" alt="logo" />
        <p>
          Edit <code>src/App.js</code> and save to reload.
        </p>
        <a
          className="App-link"
          href="https://reactjs.org"
          target="_blank"
          rel="noopener noreferrer"
        >
          Learn React
        </a>
        </header>
      </div>
    );
  }

  export default App
```

Браузер автоматически отображает любые правки в этом файле при исполняющемся приложении, обновляя страницу.

После завершения разработки приложения, чтобы создать рабочий код, нужно сгенерировать набор статических файлов, которые затем можно разместить на обычном веб-сервере. Эта задача решается с помощью команды `build`:

```
$ npm run build
```

В результате исполнения этой команды создается папка build, в которой публикуется набор статических файлов (рис. 1.2).

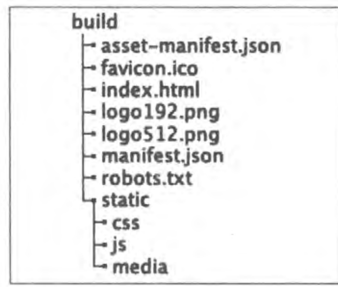


Рис. 1.2. Содержимое папки build

Многие из файлов в папке build копируются из папки public/. Код для приложения транспилировался[1] в код JavaScript, совместимый с браузерами, и был сохранен в одном или нескольких файлах в папке static/js.

Используемые приложением таблицы стилей собраны вместе и сохранены в папке static/css. К названиям некоторых из этих файлов добавлены идентификационные хеши для того, чтобы после выкладки приложения браузеры могли загружать самый свежий код, а не старые кешированные версии.

Обсуждение

Средство `create-react-app` не позволяет создавать новые приложения, но также служит платформой для поддержания приложений React в актуальном состоянии, с самыми последними версиями инструментов и библиотек. Библиотеку сценариев react можно обновить, как любую другую библиотеку, изменив номер версии и выполнив повторно команду `npm install`. Пользователю не требуется самостоятельно управлять списком подключаемых модулей (plugin) компилятора Babel или библиотек средства PostCSS или поддерживать в актуальном состоянии файл webpack.config.js. Все эти задачи выполняются библиотекой `react-scripts`.

Конечно же, все конфигурационные данные никуда не делись, но они спрятаны глубоко внутри папки библиотеки `react-scripts`, а именно в подпапке config. В этой же подпапке также находится файл webpack.config.js, который содержит информацию обо всех настройках компилятора Babel и загрузчиках файлов, которые будут необходимы приложению. Поскольку React Scripts является библиотекой, ее можно обновлять как любую другую зависимость.

Но если в дальнейшем вы решите, что больше не нуждаетесь в помощи для управления настройками своего приложения, то можете удалить свое приложение из среды `create-react-app`, выполнив команду `eject`.

```
$ npm run eject
```

После выполнения этой команды вы получите полный контроль над настройками своего приложения. Но имейте в виду, что если затем вы обнаружите, что не совсем готовы к этому, то пути назад нет. Поэтому прежде чем удалять свое приложение из среды `create-react-app`, нужно хорошенько подумать. В некоторых случаях не обязательно задавать команду `eject`, чтобы получить возможность выполнять требуемые настройки, т. к. они могут быть доступны в среде `create-react-app`. Например, разработчики часто выполняют команду `eject`, чтобы получить возможность работать с TypeScript. Но теперь то же самое можно сделать, назначив опцию `--template typescript` при создании своего приложения с помощью средства `create-react-app`.

Другой распространенной причиной выхода из среды `create-react-app` была необходимость предоставления веб-служб через прокси-сервер. Приложениям React часто

---

[1] Transpiler — транспилер (транспайлер, транспилятор), компилятор, преобразующий исходный код на одном языке программирования в исходный код на другом язык программирования. — *Прим. пер.*

требуется подключаться к какому-то отдельному серверному API-процессу. Раньше для этого разработчики настраивали сборщик пакетов Webpack для предоставления доступа к удаленному серверу через прокси-сервер на локальном сервере для разработки. Теперь же вместо этого прокси-сервер можно указывать в файле package.json:

```
"proxy": "http://myapiserver",
```

Если теперь код обращается к какому-либо адресу URL, который сервер не может найти на локальной машине (например, **/api/thing**), библиотека `react-scripts` автоматически направляет эти запросы на прокси-сервер по адресу **http://myapiserver/api/thing**.

Прежде чем удалять разрабатываемое приложение из среды `create-react-app`, попробуйте выполнить требуемые настройки, используя возможности этой среды Дополнительную информацию по этому вопросу можно найти в документации среды по адресу **https://oreil.ly/99led**

Исходный код данного рецепта на языке JavaScript или TypeScript можно загрузить на веб-сайте GitHub по адресу **https://oreil.ly/UK0dZ** или **https://oreil.ly/oOSo9** соответственно.

1.2. Создание приложений с обширным информационным наполнением посредством Gatsby

ЗАДАЧА

Веб-сайты с обширным информационным наполнением, такие как, например, блоги или магазины, должны предоставлять большие объемы сложного содержимого эффективным образом. Средство `create-react-app` не подходит для создания веб-сайтов такого типа, поскольку оно предоставляет все содержимое в виде одного большого пакета кода JavaScript, который браузер должен полностью загрузить, прежде чем что-либо будет отображено.

РЕШЕНИЕ

Для создания веб-сайтов с обширным информационным наполнением можно использовать средство Gatsby.

Это средство предназначено для загрузки, преобразования и доставки содержимого наиболее эффективным образом. Оно может создавать статические версии веб-страниц, в результате время отклика веб-сайтов, созданных с помощью этого средства, часто значительно меньше, чем сайтов, созданных на основе средства `create-react-app`.

Для Gatsby предоставляется большое число подключаемых модулей, которые могут эффективно загружать и преобразовывать данные из хранилищ локальных данных,

источников GraphQL или сторонних систем управления контентом, таких как WordPress.

Средство Gatsby можно установить глобально или же вызывать с помощью команды npx:

```
$ npx gatsby new my-app
```

В данном случае исполнение команды `gatsby new` создает папку с новым проектом, называющуюся my-app. При первом исполнении этой команды выводится запрос, какой диспетчер пакетов выбрать: `yarn` или `npm`.

Чтобы активизировать созданное приложение, нужно перейти в его папку и запустить его на исполнение в режиме разработки:

```
$ cd my-app
$ npm run develop
```

После этого приложение можно открыть в браузере по адресу **http://localhost:8000**, как показано на рис. 1.3.

Проекты Gatsby имеют прямолинейную файловую структуру, как можно видеть на рис. 1.4.

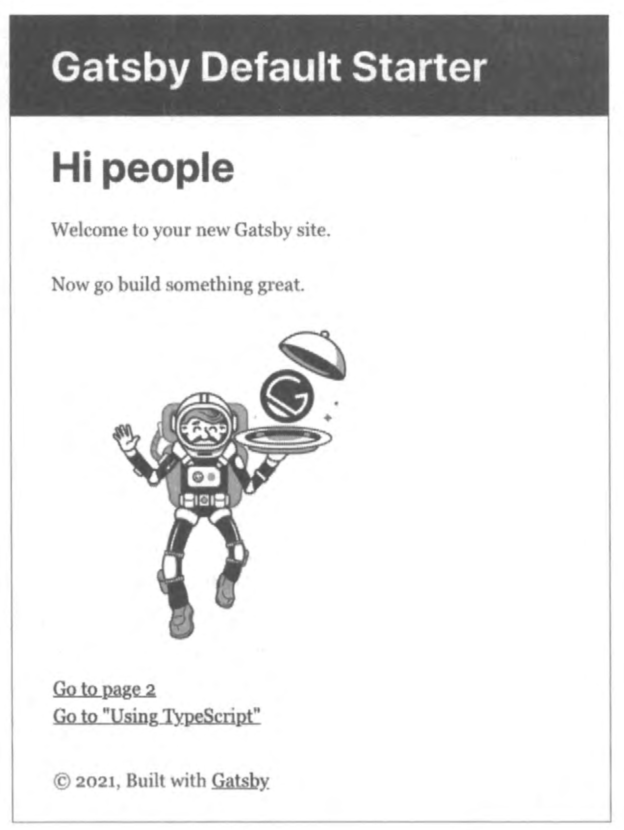


Рис. 1.3. Домашняя страница нового приложения Gatsby, открываемая по адресу **http://localhost:8000**

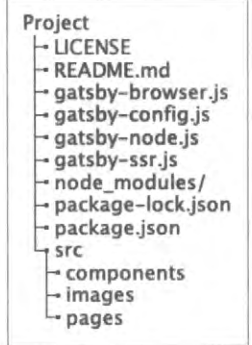


Рис. 1.4. Файловая структура проекта Gatsby

Основная часть кода приложения находится в папке src. Каждая страница приложения Gatsby имеет свой отдельный компонент React. В листинге 1.3 приведен код файла index.js домашней страницы стандартного приложения.

Листинг 1.3. Код файла index.js домашней страницы стандартного приложения Gatsby

```
import * as React from "react"
import { Link } from "gatsby"
import { StaticImage } from "gatsby-plugin-image"
import Layout from "../components/layout"
import Seo from "../components/seo"
const IndexPage = () => (
   <Layout>
      <Seo title="Home" />
      <h1>Hi people</h1>
      <p>Welcome to your new Gatsby site.</p>
      <p>Now go build something great.</p>
      <StaticImage
         src="../images/gatsby-astronaut.png"
         width={300}
         quality={95}
         formats={["AUTO", "WEBP", "AVIF"]}
         alt="A Gatsby astronaut"
         style={{ marginBottom: `1.45rem` }}
      />
      <p>
         <Link to="/page-2/">Go to page 2</Link> <br />
         <Link to="/using-typescript/">Go to "Using TypeScript"</Link>
      </p>
   </Layout>
)
export default IndexPage
```

Создавать маршрут для страницы нет необходимости, поскольку маршрут назначается автоматически каждому страничному компоненту. Например, страница с путем src/pages/using-typescript.tsx автоматически предоставляется по адресу /using-typescript[2]. Данный подход обладает многочисленными достоинствами. Прежде всего, разработчику не нужно управлять маршрутами страниц, что может быть сопряжено со значительными трудностями в случае их большого количества. Кроме того, Gatsby может предоставлять страницы намного быстрее. Чтобы понять почему, сначала разберемся, как создать сборку промышленного кода из приложения Gatsby.

[2] Да, это означает, что Gatsby поддерживает TypeScript.

Для этого нужно остановить сервер для разработки[3], а затем выполнить следующую команду:

```
$ npm run build
```

Эта команда запускает на исполнение команду `gatsby build`, которая создает папку public, содержащую все богатство Gatsby. Для каждой страницы в ней есть два файла. Первый файл содержит сгенерированный код JavaScript. Например:

```
1389 06:48 component—src-pages-using-typescript-tsx-93b78cfadc08d7d203c6.js
```

Можно видеть, что объем кода для файла using-typescript.tsx составляет всего лишь 1,389 байта, чего в комбинации с базовым фреймворком как раз достаточно, чтобы создать страницу. Иными словами, это не сценарий типа "все свое ношу с собой", которые используются в проектах `create-react-app`.

Далее, для каждой страницы есть подпапка, содержащая файл HTML. Для файла using-typescript.tsx этот файл с учетом полного пути от корневой папки называется public/using-typescript/index.html и содержит сгенерированную статическую версию веб-страницы. В частности, он содержит HTML-код, который в противном случае элемент, использующий файл using-typescript.tsx, генерировал бы динамически. В конце страницы этот код загружает версию JavaScript страницы, чтобы сгенерировать динамическое содержимое.

Такая файловая структура означает, что веб-страницы приложений Gatsby загружаются так же быстро, как и статические страницы. А с помощью встроенной библиотеки `react-helmet` можно также сгенерировать заголовочные теги `<meta/>`, содержащие информацию о дополнительных особенностях сайта. Оба этих свойства очень полезны при оптимизации веб-сайта для поднятия его позиций в результатах выдачи поисковых систем (SEO).

Обсуждение

Для добавления содержимого в приложения Gatsby можно использовать "безголовую" систему управления контентом[4], службу GraphQL, статический источник информации или какой-либо другой источник. К счастью, для Gatsby предлагается множество подключаемых модулей, посредством которых можно подключать источники данных к приложению, а затем преобразовывать содержимое из других форматов, например Markdown, в код HTML.

Полный список доступных подключаемых модулей предоставляется на веб-сайте Gatsby (**https://oreil.ly/9GwLv**).

В большинстве случаев рекомендуется выбирать требуемые подключаемые модули при создании проекта. Gatsby также поддерживает стартовые шаблоны, предостав-

---

[3] В большинстве операционных систем это можно сделать, нажав комбинацию клавиш <Ctrl>+<C>.

[4] Англ. *headless content management system*. Только серверная часть системы управления контентом (CMS), построенная с нуля как репозиторий контента, что делает контент доступным через API для отображения на любом устройстве. Термин "безголовый" происходит от концепции отделения "головы" (внешний интерфейс, т. е. веб-сайт) от "тела" (задний конец, т. е. репозиторий контента). — *Прим. пер.*

ляющие начальную структуру и настройки приложения, что позволяет ускорить процесс разработки веб-сайтов. В созданном нами ранее приложении применен довольно простой стартовый шаблон, назначенный по умолчанию. Управление модулями, подключаемыми к приложению, осуществляется посредством файла gatsby-config.js, который находится в корневой папке приложения.

Но в доступе имеется масса стартовых шаблонов для приложений Gatsby, предварительно сконфигурированных для подключения к различным источникам данных и установленными настройками для SEO, стиля, кеширования для работы офлайн, приложений PWA (Progressive Web Application — прогрессивное веб-приложение) и т. д. Какое приложение с обширным информационным наполнением вы бы ни разрабатывали, можно найти стартовый шаблон, близкий к вашим требованиям.

С дополнительной информацией о стартовых шаблонах для приложений Gatsby можно познакомиться на веб-сайте Gatsby (**https://oreil.ly/vwUd8**). Там же можно найти и список подсказок по наиболее полезным инструментам и командам Gatsby (**https://oreil.ly/f7xbF**).

Исходный код данного рецепта можно загрузить на веб-сайте GitHub по адресу **https://oreil.ly/DzLSy**.

1.3. Создание универсальных приложений с помощью Razzle

ЗАДАЧА

Иногда в начале разработки приложения не ясно, какие основные архитектурные подходы реализовать. Создавать ли одностраничное приложение? Если производительность играет важную роль, использовать ли серверный r-код? Вам нужно будет решить, какой будет ваша платформа развертывания, а также применять ли JavaScript или TypeScript для написания своего кода.

Многие инструменты требуют принятия решения по их выбору на начальной стадии работы над проектом. Если вы позже передумаете, то в дальнейшем изменить подход к сборке и развертыванию приложения будет сложно.

РЕШЕНИЕ

Помочь отложить решение о подходе к разработке и развертыванию приложения может средство Razzle (**https://oreil.ly/3pZic**).

С помощью этого средства можно создавать универсальные приложения (**https://oreil.ly/C496O**), т. е. приложения, которые могут исполнять свой код JavaScript на сервере, на клиенте, или на той и другой стороне.

Благодаря архитектуре подключаемых модулей расширения (plugin architecture), разработчик может изменить способ разработки приложения на ее дальнейших этапах. Можно даже изменить средство разработки с React на Preact или выбрать какой-либо совершенно другой фреймворк, например Elm или Vue.

Приложение Razzle создается посредством команды `create-razzle-app`:[5]

```
$ npx create-razzle-app my-app
```

В результате исполнения этой команды создается папка с указанным названием (в данном примере это my-app), в которой и размещается новый проект Razzle. Сервер для разработки запускается исполнением следующих двух команд:

```
$ cd my-app
$ npm run start
```

В результате исполнения этих команд динамически генерируется код как для клиентской версии приложения, так и для серверной и на порту 3000 запускается сервер (рис. 1.5).

Рис. 1.5. Домашняя страница нового приложения Razzle, открываемая по адресу **http://localhost:3000**

Сборка окончательной версии приложения осуществляется с помощью следующей команды:

```
$ npm run build
```

В отличие от приложений `create-react-app`, при сборке приложений Razzle создается не только код клиента, но также и рабочий сервер Node. Созданная таким образом сборка приложения помещается в подпапке build папки проекта. При исполнении серверного кода будет генерироваться статический код для клиента. Рабочий веб-сервер запускается исполнением сценария в файле build/server по следующей команде:

```
$ npm run start:prod
```

Рабочий веб-сервер можно использовать всюду, где доступен Node.js.

Как сервер, так и клиент исполняет один и тот же код, что делает приложения Razzle универсальными. Это возможно благодаря наличию разных точек входа для

[5] Название команды преднамеренно сделано похожим на команду `create-react-app`. Разработчик Razzle Джаред Палмер (Jared Palmer) считает средство `create-react-app` одним из источников своего вдохновения для его создания.

клиента и сервера. Сервер исполняет код в файле src/server.js, а браузер — код в файле src/client.js. Код в каждом из этих файлов затем прорисовывает то же самое приложение, исполняя код в файле src/App.js.

Приложение Razzle можно исполнять как одностраничное приложение. Для этого нужно удалить файлы src/index.js и src/server.js, а затем создать файл index.html в папке public. В этом файле нужно создать элемент `<div/>` с идентификатором `id = "root"`, а затем выполнить повторную сборку приложения, запустив следующую команду:

```
$ node_modules/.bin/razzle build --type=spa
```

Чтобы создавать одностраничные приложения с самого начала, нужно добавить опцию `-type=spa` в сценарии `start` и `build` в файле package.json.

В результате в папке build/public будет создано законченное одностраничное приложение, которое можно развернуть на любом веб-сервере.

Обсуждение

Такая гибкость средства Razzle возможна благодаря его структуре, состоящей из набора конфигурируемых подключаемых модулей. Каждый такой модуль представляет собой функцию высшего порядка, которая принимает конфигурацию Webpack и возвращает ее модифицированную версию. Например, один модуль может транспилировать код TypeScript, а другой — выполнять сборку библиотек React.

Чтобы переключить приложение на фреймворк Vue, нужно просто установить соответствующие модули.

Полный список доступных подключаемых модулей предоставляется на веб-сайте Razzle (**https://oreil.ly/UXwPv**).

Исходный код данного рецепта можно загрузить на веб-сайте GitHub по адресу **https://oreil.ly/ rBR9r**.

1.4. Создание серверного и клиентского кода посредством Next.js

ЗАДАЧА

Средство React создает клиентский код, даже если этот код генерируется на сервере. Но иногда при сравнительно небольшом объеме кода для API-интерфейса с этим кодом может быть предпочтительнее работать как с частью того же приложения React.

РЕШЕНИЕ

Сгенерировать код приложений React и серверный код можно с помощью фреймворка Next.js. Концевые точки API-интерфейса и клиентских страниц используют соглашения о маршрутизации по умолчанию, что упрощает их создание и развертывание, в отличие от управления маршрутизацией вручную. Полная информация о фреймворке Next.js предоставляется на его веб-сайте (**https://nextjs.org**).

Приложение Next.js создается при помощи следующей команды:

```
$ npx create-next-app my-app
```

Если установлен диспетчер пакетов `yarn`, то в приложении будет использован этот диспетчер. Но можно также и задать диспетчер пакетов `npm`, указав флаг `--user-npm`:

```
$ npx create-next-app --use-npm my-app
```

В результате исполнения данной команды будет создана указанная подпапка (my-app в данном случае), в которой будет размещено новое приложение Next.jc. Чтобы запустить созданное приложение, нужно перейти в его папку и запустить его на исполнение в режиме разработки (рис. 1.6):

```
$ cd my-app
$ npm run dev
```

Средство Next.js позволяет создавать страницы, не беспокоясь при этом об управлении конфигурацией маршрутизации. Таким образом, добавленный в папку pages компонент сразу же становится доступным для раздачи сервером. Например, компонент pages/index.js создает домашнюю страницу стандартного приложения.

Рис. 1.6. Страница приложения Next.js, открываемая по адресу **http://localhost:3000**

Данный способ похож на подход, применяемый в Gatsby[6], но отличается добавлением серверного кода.

Приложения Next.js обычно содержат определенный объем серверного кода API-интерфейса, чего, как правило, нет в приложениях React, которые часто собираются отдельно от серверного кода. В частности, в подпапке pages/api можно найти пример концевой точки сервера, называющейся hello.js. Содержимое этого файла приведено в листинге 1.4.

Листинг 1.4. Содержимое файла концевой точки сервера

```
// Next.js API route support: https://nextjs.org/docs/api-routes/introduction

export default (req, res) => {
   res.status(200).json({ name: 'John Doe' })
}
```

Маршрутизация, которая подключает код к концевой точке api/hello, выполняется автоматически.

Средство Next.js транспилирует код приложения и сохраняет его в скрытой папке .next, которая в дальнейшем может быть развернута на службе, например на платформе Vercel (**https://vercel.com**) самого Next.js.

При желании можно сгенерировать статическую сборку приложения, выполнив следующую команду:

```
$ node_modules/.bin/next export
```

Созданный в результате исполнения команды код клиента сохраняется в папке out. Команда преобразовывает каждую страницу приложения в статический HTML-файл, что ускоряет ее загрузку в браузере. В конце страницы загружается версия JavaScript, чтобы сгенерировать динамическое содержимое.

Созданные посредством команды `export` статические версии приложений Next.js не содержат никакого кода серверного API-интерфейса.

Средство Next.js поддерживает большое количество опций получения данных, что позволяет получать данные из статического содержимого или из "безголовых" систем управления контентом (**https://oreil.ly/Xmia8**).

Обсуждение

Средства Next.js и Gatsby во многих аспектах схожи. Их главные отличительные особенности — скорость предоставления содержимого при небольшом объеме

---

[6] См. *раздел 1.2.*

конфигурирования. Скорее всего, Next.js будет наиболее полезным для команд разработчиков, которые имеют дело с очень небольшим объемом серверного кода.

Исходный код данного рецепта можно загрузить на веб-сайте GitHub по адресу **https://oreil.ly/9gbJs**.

1.5. Создание крошечных приложений посредством Preact

ЗАДАЧА

Объем приложений React может быть достаточно большим. Даже простое приложение React может с легкостью транспилироваться в сборки кода JavaScript размером в несколько сотен килобайт. А в данном случае требуется создать приложение с возможностями платформы React, но намного меньшего размера.

РЕШЕНИЕ

Для создания приложения с возможностями React, не расплачиваясь за это сборкой кода JavaScript большого объема, можно использовать средство Preact.

Несмотря на очень похожие названия, средство Preact — это не React. Это отдельная библиотека, которая предназначена для создания приложений, как можно больше похожих на приложения React, но намного меньшего размера.

Большой размер приложений React объясняется принципом работы этого фреймворка. Компоненты приложений React не создают элементы непосредственно в модели DOM (Document Object Model — объектная модель документов) браузера, а в виртуальной модели DOM, а затем обновляют настоящую модель DOM через короткие интервалы времени. Такой подход позволяет быстро прорисовывать базовую модель DOM, поскольку настоящую модель DOM нужно обновлять только при наличии реальных изменений. Но это имеет свою отрицательную сторону в виде большого объема кода, необходимого для поддержания виртуальной модели DOM приложений React в актуальном состоянии. Поскольку фреймворку React необходимо содержать всю синтетическую модель событий, аналогичную соответствующей модели браузера, приложения React могут иметь большой размер и занимать некоторое время для загрузки.

Один из способов нейтрализации указанного недостатка состоит в отрисовке страниц на сервере (SSR — Server-Side Rendering), но подобный метод может оказаться сложным для настройки[7]. Иногда нужно лишь загрузить небольшой объем кода, для таких случаев и был создан фреймворк Preact.

Хотя библиотека Preact похожа на библиотеку React, она имеет небольшой размер. На момент подготовки материала данной книге размер основной библиотеки Preact

[7] См. *разделы 1.2* и *1.3*.

составляет около 4 Кбайт, что позволяет добавлять в веб-страницы возможности в стиле React при объеме кода немногим больше, чем обычный браузерный код JavaScript.

React можно использовать или как небольшую библиотеку JavaScript, включенную в веб-страницу (подход без поддержки инструментов), или как полнофункциональное приложение JavaScript.

Подход без применения инструментов предоставляет базовые возможности. Основная библиотека Preact не поддерживает расширение JSX, что означает отсутствие поддержки компилятора Babel, что, в свою очередь, не позволяет реализовать современные возможности языка JavaScript. В листинге 1.5 приведен пример кода страницы с использованием библиотеки Preact.

Листинг 1.5. Код страницы с использованием библиотеки Preact

```
<html>
   <head>
      <title>No Tools!</title>
      <script src="https://unpkg.com/preact?umd"></script>
   </head>
   <body>
      <h1>No Tools Preact App!</h1>
       <div id="root"></div>
       <script>
         var h = window.preact.h;
         var render = window.preact.render;

         var mount = document.getElementById('root');
         render(
            h('button',
               {
                  onClick: function() {
                     render(h('div', null, 'Hello'), mount);
                  }
               },
               'Click!'),
            mount
         );
      </script>
   </body>
</html>
```

Это приложение устанавливается в элементе `<div/>` с идентификатором `id="root"`, где оно будет отображать кнопку. При нажатии этой кнопки содержимое элемента `div` заменяется строкой "Hello". Более простое приложение Preact вряд ли можно придумать.

Но настоящие приложения Preact редко разрабатываются таким образом. В действительности они обычно создаются на основе простой цепочки сборок, которая как минимум поддерживает современный язык JavaScript.

Средство Preact поддерживает весь диапазон приложений JavaScript. С другой стороны, полноценное приложение Preact можно создать с помощью средства Preact CLI.

Этот инструмент для создания проектов Preact аналогичен инструментам типа `create-react-app`. Для создания приложения Preact посредством этого инструмента предназначена следующая команда:

```
$ npx preact-cli create default my-app
```

Данная команда использует для создания приложения шаблон по умолчанию. Но доступны и другие шаблоны для создания проектов, например, с компонентами Material или языком TypeScript. Дополнительную информацию см. на странице GitHub фреймворка Preact (**https://oreil.ly/lVQua**).

В результате исполнения приведенной команды будет создана указанная подпапка (my-app в данном случае), в которой будет размещено новое приложение Preact. Приложение запускается следующей последовательностью команд:

```
$ cd my-app
$ npm run dev
```

В результате запустится сервер на порту 8080, раздающий страницу, показанную на рис. 1.7.

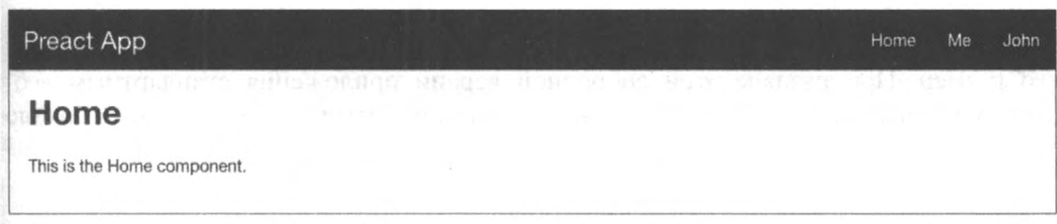


Рис. 1.7. Страница приложения Preact

Сервер создает веб-страницу, которая вызывает пакет JavaScript, собранный из кода в файле src/index.js.

В результате получается полномасштабное приложение наподобие приложения Preact. Например, код в компоненте Home (src/routes/home/index.js) выглядит во многом как код приложения React с полной поддержкой языка JSX (листинг 1.6).

Листинг 1.6. Код приложения Preact, созданного при помощи Preact CLI

```
import { h } from 'preact';
import style from './style.css';
const Home = () => (
  <div class={style.home}>
```

```
      <h1>Home</h1>
      <p>This is the Home component.</p>
   </div>
);
export default Home;
```

Единственное, чем этот компонент значительно отличается от стандартного компонента React, так это импортированием функции `h` из библиотеки `preact`, вместо импортирования `React` из библиотеки `react`.

Код JSX в приложениях Preact преобразуется в последовательность вызовов функции `h`, чем и объясняется надобность в ее импортировании. По этой же причине приложения, созданные посредством версий `create-react-app` более ранних, чем версия 17, также требовали импортирования объекта `react`. Начиная с версии 17, в `create-react-app` осуществляется преобразование в JSX (**https://oreil.ly/HOwS9**), что избавляет от необходимости импортировать `react` каждый раз. Можно надеяться, что в будущих версиях Preact будет осуществлено подобное изменение.

Но размер приложения увеличился, составляя теперь чуть свыше 300 Кбайт. Это довольно много, но мы все еще находимся в режиме разработки. Чтобы увидеть настоящую мощь Preact, остановим сервер для разработки (нажав клавиши <Ctrl>+<C>), а затем выполним сборку приложения, исполнив следующую команду:

```
$ npm run build
```

В результате будет создана папка build, в которой появится сгенерированная статическая версия приложения. Первым положительным результатом этого будет создание статической копии домашней страницы, которая станет быстро отображаться. А второй результат — удаление всего лишнего кода из приложения, уменьшая его размер. При раздаче этой собранной версии приложения стандартным веб-сервером браузер после отображения страницы загрузит всего лишь около 50–60 Кбайт.

Обсуждение

Средство Preact предоставляет замечательные возможности. Несмотря на значительно иной принцип работы, чем React, оно обладает практически такой же мощью, будучи намного меньшим по размеру. И тот факт, что с его помощью можно создавать проекты в диапазоне от примитивного встроенного кода до полномасштабных одностраничных приложений, означает, что использование его для собственной разработки вполне достойно рассмотрения, если объем кода критически важен.

Дополнительную информацию о фреймворке Preact можно найти на его веб-сайте (**https://preactjs.com**).

Исходный код примера приложения без включения инструментов и код примера полномасштабного приложения можно загрузить на веб-сайте GitHub по адресу **https://oreil.ly/N9PKf** и **https://oreil.ly/F0tW9** соответственно.

Приложения Preact можно сделать еще более похожими на приложения React с помощью библиотеки `preact-compat` (**https://oreil.ly/3YXOv**).

Наконец, библиотека InfernoJS (**https://infernojs.org**) предоставляет еще одну альтернативу React наподобие подхода Preact.

1.6. Создание библиотек посредством набора инструментов *nwb*

ЗАДАЧА

В крупных организациях часто одновременно разрабатывается несколько приложений React. Например, консалтинговая компания может создавать несколько приложений для разных организаций. А компания по разработке программного обеспечения может разрабатывать разные приложения с одинаковым оформлением или функциональностью, для чего, скорее всего, потребуются общие компоненты в нескольких приложениях.

При создании компонентного проекта необходимо сформировать структуру папок, выбрать набор инструментов и набор языковых возможностей, а также реализовать сборочную цепочку, которая может собрать компонент в развертываемый формат. Этот процесс может быть таким же трудоемким, как и ручное создание всего приложения React.

РЕШЕНИЕ

С помощью набора инструментов `nwb` можно создавать как полномасштабные приложения, так и отдельные компоненты React. Он также позволяет создавать компоненты для использования в проектах Preact и InfernoJS, но здесь мы будем концентрироваться на рассмотрении компонентов React.

Чтобы использовать набор инструментов `nwb` в проектах React, нужно установить его глобально:

```
$ npm install -g nwb
```

После этого можно будет создавать новые проекты, исполняя команду `nwb`:

```
$ nwb new react-component my-component
```

Вместо одного компонента можно создать полноценное приложение `mwb` во фреймворке `react`, `preact` или `inferno`, заменив опцию `react-component` на `react-app`, `preact-act` или `inferno-app` соответственно. Также можно создать и обычный проект JavaScript, указав опцию `vanilla-app`.

После ввода этой команды система задаст несколько вопросов о том, какой тип библиотеки выбрать. В частности, будет задан вопрос, хотите ли вы создать сборку модулей сценариев ECMAScript (ES):

```
Creating a react-component project...
(Создается проект компонента react...)
? Do you want to create an ES modules build? (Y/n)
(? Хотите ли создать сборку модулей ES? (Да/нет))
```

Данная опция включает в приложение оператор экспорта, с помощью которого сборщик пакетов Webpack может решить, нужно ли включать компонент в клиентское приложение. Также задается вопрос, хотите ли вы создать определение UMD (Universal Module Definition — универсальное определение модуля):

```
? Do you want to create a UMD build? (y/N)
 (Хотите ли вы создать сборку с определением UMD? (Да/нет))
```

Такое определение будет полезным в том случае, когда разрабатываемый компонент нужно будет включить в элемент `<script/>` веб-страницы. Для нашего примера мы не будем создавать сборку с определением UMD.

Получив ответы на все вопросы, средство `nwb` создаст указанную в команде подпапку (my-component в данном случае), а в ней — проект компонента `nwb`. Проект примера состоит из простого приложения-обертки, которое запускается исполнением следующих двух команд:

```
$ cd my-component
$ npm run start
```

В результате запустится сервер на порту 3000, раздающий страницу, показанную на рис. 1.8.

Рис. 1.8. Страница компонента `nwb`

Приложение содержит один компонент, который определяется в файле src/index.js (листинг 1.7).

Листинг 1.7. Исходный код компонента проекта `nwb`

```
import React, { Component } from 'react'
export default class extends Component {
   render() {
      return (
         <div>
            <h2>Welcome to React components</h2>
         </div>
      )
   }
}
```

Теперь можно продолжить разработку компонента, как любого другого проекта React. Сборка компонента для развертывания осуществляется следующей командой:

```
$ npm run build
```

Сборка будет находиться в файле lib/index.js, который можно разместить в хранилище для использования в других проектах.

Обсуждение

Дополнительная информация по созданию компонентов `nwb` предоставляется в руководстве `nwb` по разработке компонентов и библиотек (**https://oreil.ly/XHrQa**).

Исходный код данного рецепта можно загрузить на веб-сайте GitHub по адресу **https://oreil.ly/P4Xzj**.

1.7. Добавление React в код Rails посредством Webpacker

ЗАДАЧА

Фреймворк Rails был создан до того, как интерактивные приложения JavaScript набрали популярность. Приложения Rails следуют более традиционной модели разработки веб-приложений, в которой страницы HTML создаются в ответ на запросы браузера. Но иногда может потребоваться добавить в приложение Rails элементы с бо́льшими интерактивными возможностями.

РЕШЕНИЕ

Приложения React можно вставить в веб-страницы Rail с помощью библиотеки Webpacker. Для начала создадим приложение Rails, содержащее библиотеку Webpacker:

```
$ rails new my-app --webpack=react
```

В результате будет сформирована подпапка my-app, в которой будет создано приложение Rails, уже сконфигурированное для работы с сервером Webpacker. Прежде чем запускать это приложение, перейдем в его папку и создадим пример страницы/контроллера:

```
$ cd my-app
$ rails generate controller Example index
```

Созданная таким образом страница app/views/example/index.html.erb имеет следующее содержимое:

```
<h1>Example#index</h1>
<p>Find me in app/views/example/index.html.erb</p>
```

Теперь нам нужно создать небольшое приложение React, чтобы вставить его в эту страницу. В Rails приложения Webpacker вставляются в виде небольших пакетов

кода JavaScript в коде Rails. Мы создадим новый пакет, содержащий простой компонент счетчика, в файле app/javascript/packs/counter.js (листинг 1.8).

Листинг 1.8. Код компонента счетчика

```
import React, { useState } from 'react'
import ReactDOM from 'react-dom'
const Counter = (props) => {
   const [count, setCount] = useState(0)
   return (
      <div className="Counter">
         You have clicked the button {count} times.
         <button onClick={() => setCount((c) => c + 1)}>Click!</button>
      </div>
   )
}
document.addEventListener('DOMContentLoaded', () => {
   ReactDOM.render(
      <Counter />,
      document.body.appendChild(document.createElement('div'))
   )
})
```

При каждом нажатии кнопки приложение обновляет значения счетчика.

Теперь вставим этот пакет в нашу веб-страницу примера, добавив в стандартную страницу одну строку кода (выделена полужирным шрифтом):

```
<h1>Example#index</h1>
<p>Find me in app/views/example/index.html.erb</p>
<%= javascript_pack_tag 'counter' %>
```

Далее запустим сервер Rails:

```
$ rails server
```

На момент работы над материалом данной книги для запуска сервера Rails нужно, чтобы был установлен диспетчер пакетов `yarn`. Установить его глобально можно, выполнив команду `npm install -g yarn`.

Теперь введем в адресную строку браузера адрес страницы **http://localhost:3000/example/index.html**, и она должна открыться в браузере (рис. 1.9).

Example#index

Find me in app/views/example/index.html.erb

You have clicked the button 3 times Click!

Рис. 1.9. Приложение React, встроенное в страницу Rails

Обсуждение

Как вы уже, наверное, догадались, за кулисами браузера библиотека Webpacker преобразовывает приложение с помощью копии сборщика пакетов Webpack. Настройки можно задавать, редактируя конфигурационный файл app/config/webpacker.yml.

Библиотека Webpacker используется попутно с кодом Rails, а не вместо него. Такой подход следует рассмотреть, когда приложению Rails необходимо придать немного дополнительной интерактивности.

Дополнительная информация о библиотеке Webpacker предоставляется на ее веб-сайте GitHub (**https://oreil.ly/aYZ0h**).

Исходный код данного рецепта можно загрузить на веб-сайте GitHub по адресу **https://oreil.ly/H3qlF**.

1.8. Создание пользовательских элементов посредством Preact

ЗАДАЧА

В ряде случаев возникают обстоятельства, затрудняющие добавление кода React в существующее содержимое. Например, в некоторых конфигурациях систем CMS пользователям не разрешается вставлять дополнительный код JavaScript в код страницы. В таких случаях было бы полезным иметь какой-либо стандартный способ безопасно добавлять код JavaScript в страницу.

РЕШЕНИЕ

Применение пользовательских элементов — стандартный способ создания новых элементов HTML для веб-страниц. По сути, они расширяют язык HTML, делая доступными для пользователя дополнительные элементы.

В этом рецепте мы рассмотрим, как с помощью облегченного фреймворка Preact создавать пользовательские элементы, которые затем можно разместить на стороннем сервере.

Начнем с создания нового приложения Preact. Это приложение будет предоставлять пользовательский элемент, который можно будет использовать в других веб-страницах[8]:

```
$ preact create default my-element
```

Затем перейдем в папку созданного приложения и добавим к проекту библиотеку `preact-custon-element`:

```
$ cd my-element
$ npm install preact-custom-element
```

[8] Дополнительную информацию по созданию приложений Preact см. в *разделе 1.5*.

Эта библиотека позволит нам регистрировать в браузере новые пользовательские HTML-элементы.

Теперь нам нужно отредактировать файл src/index.js проекта, чтобы он регистрировал новый пользовательский элемент, который мы назовем components/Converter/index.js. Для этого добавим в данный файл следующий код:

```
import register from 'preact-custom-element'
import Converter from './components/Converter'
register(Converter, 'x-converter', ['currency'])
```

Метод `register` указывает браузеру, что нужно создать новый пользовательский HTML-элемент с названием `<x-converter/>`, имеющий одно свойство, называющееся `currency`. Это свойство определяется в файле src/components/Converter/index.js (листинг 1.9).

Листинг 1.9. Код для определения свойства `currency`

```
import { h } from 'preact'
import { useEffect, useState } from 'preact/hooks'
import 'style/index.css'
const rates = { gbp: 0.81, eur: 0.92, jpy: 106.64 }
export default ({ currency = 'gbp' }) => {
   const [curr, setCurr] = useState(currency)
   const [amount, setAmount] = useState(0)
   useEffect(() => {
      setCurr(currency)
   }, [currency])
   return (
      <div className="Converter">
         <p>
         <label htmlFor="currency">Currency: </label>
         <select
            name="currency"
            value={curr}
            onChange={(evt) => setCurr(evt.target.value)}
         >
         {Object.keys(rates).map((r) => (
            <option value={r}>{r}</option>
         ))}
            </select>
         </p>
         <p className="Converter-amount">
            <label htmlFor="amount">Amount: </label>
            <input
               name="amount"
               size={8}
               type="number"
```

```
            value={amount}
            onInput={(evt) => setAmount(parseFloat(evt.target.value))}
          />
        </p>
        <p>
          Cost:
          {((amount || 0) / rates[curr]).toLocaleString('en-US', {
            style: 'currency',
            currency: 'USD',
          })}
        </p>
      </div>
    )
}
```

Согласно спецификации названия пользовательских элементов должны начинаться с прописной буквы, не содержать заглавных букв и включать дефис[9]. Таким образом обеспечивается отсутствие конфликтов с названиями стандартных элементов.

Наш компонент `Converter` представляет собой конвертер валют, использующий для простоты фиксированный курс. Если теперь запустить сервер Preact командой

```
$ npm run dev
```

то по адресу **http://localhost:8080/bundle.js** будет доступен код JavaScript для нашего пользовательского элемента.

Чтобы задействовать этот элемент, создадим обычную статическую веб-страницу, содержащую код, приведенный в листинге 1.10.

Листинг 1.10. Код HTML для веб-страницы с пользовательским элементом

```
<html>
  <head>
    <script src="https://unpkg.com/babel-polyfill/dist/polyfill.min.js">
     </script>
     <script src="https://unpkg.com/@webcomponents/webcomponentsjs">
     </script>
     <!-- Замените следующий адрес адресом своего пользовательского элемента -->
     <script type="text/javascript" src="http://localhost:8080/bundle.js">
     </script>
  </head>
  <body>
    <h1>Custom Web Element</h1>
    <div style="float: right; clear: both">
```

[9] Подробнее о пользовательских элементах см. спецификацию WHATWG (**https://oreil.ly/KOjmP**).

```
    <!-- Этот тег вставляет приложение Preact -->
      <x-converter currency="jpy"/>
    </div>
    <p>This page contains an example custom element called
      <code>&lt;x-converter/&gt;</code>,
      which is being served from a different location</p>
    </body>
</html>
```

Эта веб-страница содержит определение нашего пользовательского элемента в последнем элементе `<script/>` элемента `<head/>`. Чтобы обеспечить доступность пользовательского элемента как новым, так и старым браузерам, здесь же размещены ссылки (unpkg.com) на пару оболочек совместимости.

Теперь, когда к веб-странице подключен код пользовательского элемента, мы можем вставлять тег `<x-converter/>` в код страницы, как любой другой обычный тег HTML. В рассмотренном примере мы также передаем свойство `currency` базовому компоненту Preact.

При передаче свойств пользовательских элементов базовому компоненту их названия указываются прописными буквами, независимо от регистра букв в названии в определении.

Теперь мы можем раздавать эту страницу с пользовательским элементом с любого веб-сервера, не имеющего отношения к серверу Preact. На рис. 1.10 показано, как она должна выглядеть.

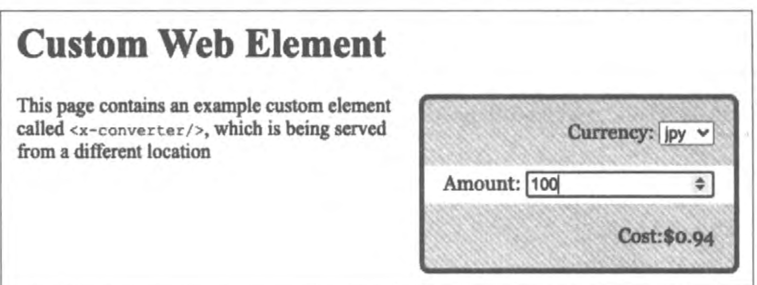


Рис. 1.10. Обычная веб-страница со вставленным в нее пользовательским элементом Preact

Обсуждение

Файл с кодом пользовательского элемента не обязательно должен находиться на том же самом сервере, что и веб-страница, в которой он используется. Это означает, что пользовательские элементы можно применять для публикации виджетов на любой веб-странице. Поэтому может быть не лишним проверять заголовок `Referer` на всех входящих запросах к компоненту, чтобы предотвратить его несанкционированное использование.

В нашем примере пользовательский элемент предоставляется сервером разработки средства Preact. Для рабочей версии элемента будет желательно создать статическую сборку компонента, которая, скорее всего, будет значительно меньшего размера[10].

Исходный код данного рецепта можно загрузить на веб-сайте GitHub по адресу **https://oreil.ly/aB7BP**.

1.9. Разработка компонентов посредством Storybook

ЗАДАЧА

Компоненты React — надежные строительные блоки для приложений React. Должным образом разработанные компоненты можно использовать в других приложениях React. Но при разработке компонента нужно приложить некоторые усилия, чтобы проверить, как он работает при всех обстоятельствах. Например, в асинхронном приложении React может отображать компонент с неопределенными свойствами. Будет ли компонент тем не менее отображен правильно или с ошибками?

Но при разработке компонентов как части сложного приложения создать все ситуации, с которыми ваш компонент должен справляться, может оказаться довольно трудной задачей.

Кроме того, специализированным разработчикам UX[11] придется тратить много времени, если им нужно будет проходить по всему приложению, чтобы просмотреть один разрабатываемый ими компонент.

Было бы неплохо каким-то образом отображать только сам компонент и передавать ему примеры наборов свойств.

РЕШЕНИЕ

Данная задача решается при помощи средства Storybook, которое позволяет отображать библиотеки компонентов в различных состояниях. Это средство можно было бы назвать галереей компонентов, но это, скорее всего, будет преуменьшением его возможностей. В действительности это инструмент для разработки компонентов.

Чтобы добавить возможности Storybook в проект, сначала создадим новое приложение React, а затем перейдем в папку созданного приложения:

```
$ npx create-react-app my-app
$ cd my-app
```

---

[10] Подробнее о средстве Preact см. *раздел 1.5*.

[11] User Experience — впечатление пользователя от взаимодействия с интерфейсом приложения. Разработчик UX отвечает за создание пользовательского интерфейса и других элементов для обеспечения наилучшего качества UX. — *Прим. пер.*

А теперь добавим Storybook к проекту:

```
$ npx sb init
```

Затем запускаем сервер Storybook:

```
$ npm run storybook
```

Сервер Storybook запустится на порту 9000, и в браузере откроется его страница (рис. 1.11). При работе со Storybook само приложение React исполнять не нужно.

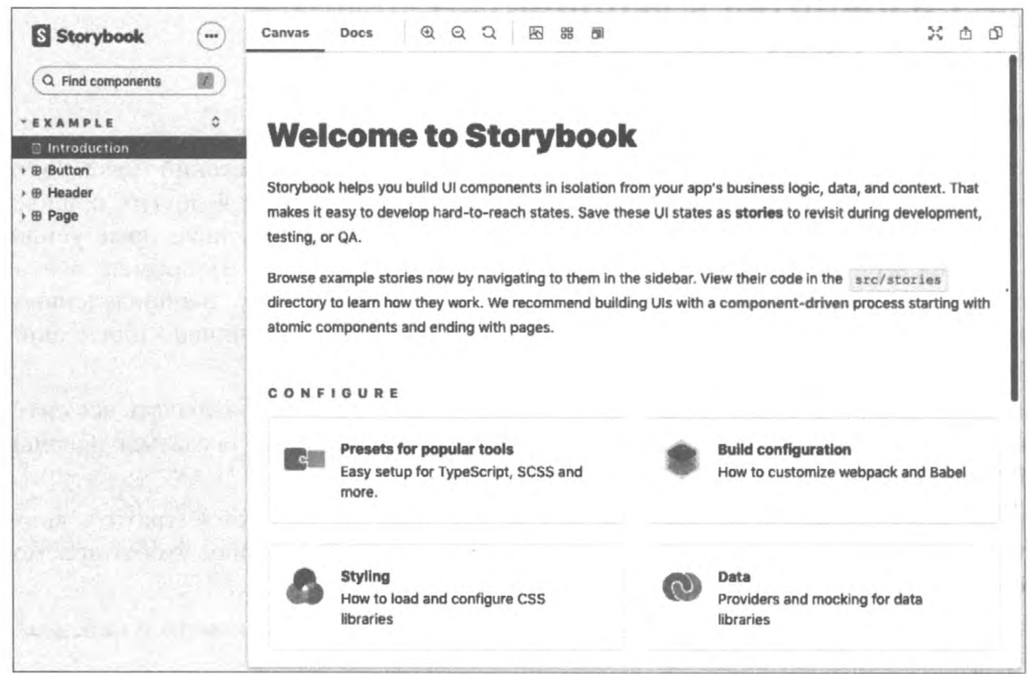


Рис. 1.11. Страница приветствия в Storybook

В Storybook прорисовка одного элемента с примером свойств называется *рассказом* (story). В установке Storybook по умолчанию образцы историй находятся в подпапке stories корневой папки проекта. Например, в листинге 1.11 приведен код файла рассказа stories/Button.stories.js:

Листинг 1.11. Код файла рассказа stories/Button.stories.js

```
import React from 'react';
import { Button } from './Button';
export default {
   title: 'Example/Button',
   component: Button,
   argTypes: {
      backgroundColor: { control: 'color' },
   },
};
```

```
const Template = (args) => <Button {...args} />;
export const Primary = Template.bind({});
Primary.args = {
   primary: true,
   label: 'Button',
};
export const Secondary = Template.bind({});
Secondary.args = {
   label: 'Button',
};
export const Large = Template.bind({});
Large.args = {
   size: 'large',
   label: 'Button',
};
export const Small = Template.bind({});
Small.args = {
   size: 'small',
   label: 'Button',
};
```

Средство Storybook отслеживает файлы с названием *.stories.js в папке с исходными файлами, и для него не имеет значения, где они находятся. Поэтому их можно сохранять в любом месте. Типичный подход — сохранять файлы рассказов в папке компонента, который они представляют. Таким образом, при копировании этой папки в другое место также копируются и рассказы как актуальная документация.

На рис. 1.12 показано, как выглядит в Storybook рассказ Button.stories.js.

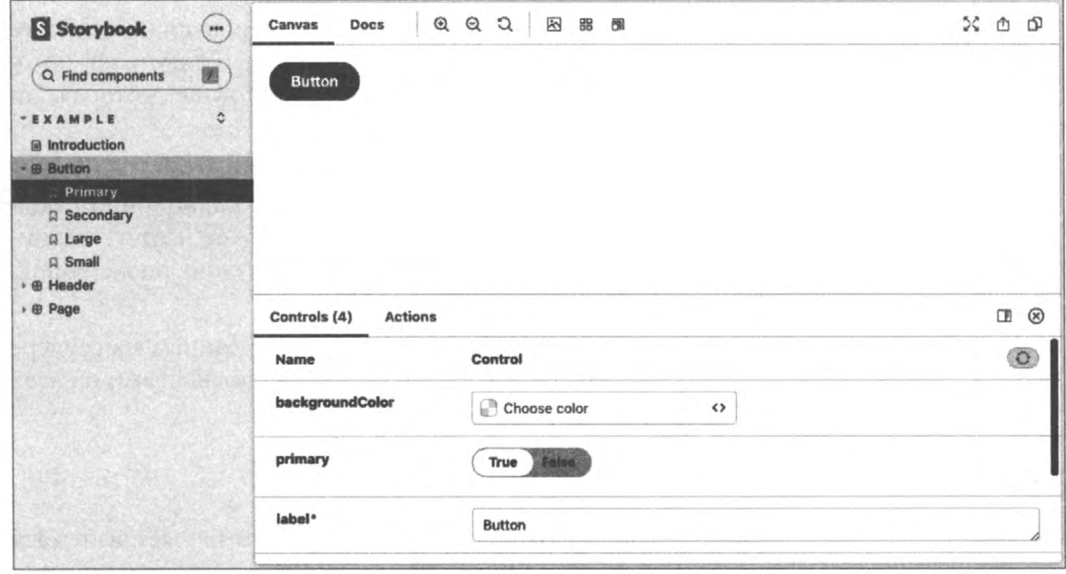


Рис. 1.12. Пример рассказа

Обсуждение

Несмотря на свою кажущуюся простоту, Storybook представляет собой эффективное средство разработки, позволяющее концентрироваться на одном компоненте за раз. Чем-то сродни визуальной проверке модулей, оно позволяет испытать компонент в нескольких возможных сценариях работы, чтобы удостовериться в его правильном поведении.

Для Storybook также предоставляется большой выбор дополнительных модулей расширений (**https://oreil.ly/3kSVa**), с помощью которых можно обеспечить дополнительные функции:

- проверка на наличие проблем с доступностью (Accessibility);
- добавление интерактивных элементов управления для задания свойств (Knobs);
- включение онлайновой документации с каждым рассказом (Docs);
- запись снимков HTML для проверки эффекта от изменений (Storyshots).

И много чего другого.

Дополнительная информация о средстве Storybook предоставляется на его веб-сайте (**https://storybook.js.org**).

Исходный код данного рецепта можно загрузить на веб-сайте GitHub по адресу **https://oreil.ly/GyxTX**.

1.10. Тестирование кода в браузере посредством Cypress

ЗАДАЧА

Большинство проектов React содержат библиотеку для тестирования. Наиболее распространенные из них — это, наверное, библиотека @testing-library/react, которая устанавливается вместе со средством `create-react-app`, и Enzyme, которая используется средством Preact.

Но наилучшее тестирование — это тестирование в настоящем браузере, со всеми вытекающими из этого дополнительными усложнениями. При традиционном подходе тестирование в браузере может быть нестабильным и требует частого технического обслуживания, т. к. при каждом обновлении браузера нужно также обновлять и его драйверы (например, ChromeDriver).

Присовокупите к этому необходимость генерирования тестовых данных на сервере бэкенда, и тестирование в браузере может быть сопряжено со сложностями на всех этапах от организации до управления.

РЕШЕНИЕ

Фреймворк для тестирования Cypress (**https://www.cypress.io**) позволяет избежать многих недостатков традиционного тестирования в браузере. Он исполняется в браузере, но не требует внешнего средства доставки веб-контента. Вместо этого

взаимодействие с браузером происходит через сетевой порт и внедряет код JavaScript для исполнения большей части тестового кода.

Чтобы рассмотреть работу этого средства, создадим приложение `create-react-app`:

```
$ npx create-react-app --use-npm my-app
```

Затем перейдем в папку созданного приложения и установим фреймворк Cypress:

```
$ cd my-app
$ npm install cypress --save-dev
```

Прежде чем запускать Cypress на исполнение, ему нужно указать, как найти наше приложение. Для этого создадим в папке приложения файл cypress.json и сохраним в нем следующий код с указанием URL-адреса нашего приложения:

```
{
  "baseUrl": "http://localhost:3000/"
}
```

Теперь можно запустить на исполнение наше основное приложение:

```
$ npm start
```

И наконец, открыть приложение Cypress:

```
$ npx cypress open
```

При первом запуске Cypress устанавливаются все требуемые для него зависимости. Теперь в папке cypress/integration создадим файл screenshot.js с кодом теста (листинг 1.12), который открывает домашнюю страницу и делает снимок кода.

Листинг 1.12. Код теста

```
describe('screenshot', () => {
  it('should be able to take a screenshot', () => {
    cy.visit('/');
    cy.screenshot('frontpage');
  });
});
```

Обратите внимание на то, что тест написан в формате Jest. После сохранения файл теста отобразится в главном окне Cypress (рис. 1.13).

Двойной щелчок мышью по файлу теста запустит его в браузере. Откроется домашняя страница приложения, и тест сохранит снимок экрана в файл cypress/screenshots/screen-shot.js/frontpage.png.

Обсуждение

В табл. 1.1 приведено несколько примеров команд Cypress.

Это всего лишь несколько команд для взаимодействия с веб-страницей. Но Cypress способен на большее. В частности, он может модифицировать код в браузере, чтобы

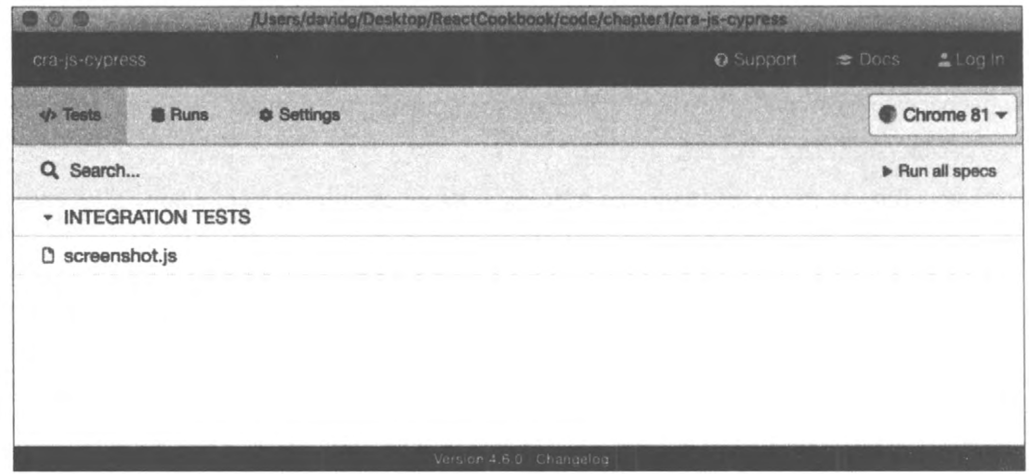


Рис. 1.13. Главное окно Cypress

Таблица 1.1. *Примеры команд фреймворка Cypress*

Команда	**Описание**
`cy.contains('Fred')`	Выполняет поиск элемента, содержащего строку "Fred"
`cy.get('.Norman').click()`	Выполняет щелчок мышью по элементу класса "Norman"
`cy.get('input').type('Hi!')`	Вводит строку "Hi!" в поле ввода
`cy.get('h1').scrollIntoView()`	Прокручивает страницу до вывода в область просмотра элемент <h1/>

изменить время (команда `cy.clock()`), файлы куки (команда `cy.setCookie()`), содержимое локального хранилища (команда `cy.clearLocalStorage()`). Но самое впечатляющее то, что он может эмулировать запросы к серверу API и ответы от него.

Для этого он модифицирует встроенные в браузер сетевые функции таким образом, что код:

```
cy.route("/api/server?*", [{some: 'Data'}])
```

перехватывает все запросы к конечной точке сервера, начинающейся с `/api/server?`, и возвращает массив JSON `[{some: 'Data'}]`.

Возможность эмулирования сетевых ответов может полностью изменить процесс разработки приложений, поскольку это разделяет этапы разработки фронтенда и бэкенда. При тестировании в браузере можно просто указать требуемые данные, без необходимости организовывать настоящий сервер и базу данных.

Дополнительная информация о фреймворке Cypress предоставляется на его веб-сайте (**https://oreil.ly/eX09t**).

Исходный код данного рецепта можно загрузить на веб-сайте GitHub по адресу **https://oreil.ly/3j8vI**.

Маршрутизация

В рецептах этой главы рассматривается использование маршрутизации в React и библиотека `react-router-dom`.

В библиотеке `react-router-dom` применяется декларативная маршрутизация, в результате чего маршруты обрабатываются, как любой другой компонент React. Хотя, в отличие от кнопок, текстовых полей и блоков текста, маршруты React не имеют визуального представления, во всех других отношениях они подобны этим элементам. Маршруты располагаются в виртуальном DOM-дереве компонентов. Они отслеживают изменения в текущем местонахождении (URL) браузера и позволяют включать и выключать части интерфейса. Таким образом одностраничные приложения создают видимость многостраничных веб-сайтов.

При правильном использовании маршрутов приложение может выглядеть как любой другой веб-сайт. Например, пользователи могут делать закладки для разделов приложения таким же образом, как и для страниц веб-сайта, например Википедии. Также они могут перемещаться по истории посещенных страниц, и интерфейс будет вести себя соответствующим образом. Если вы только начинаете изучать React, то будет очень полезным рассмотреть возможности маршрутизации более подробно.

2.1. Создание интерфейсов, используя реагирующую маршрутизацию

ЗАДАЧА

Большинство приложений предназначено как для мобильных, так и для настольных компьютеров, поэтому качественные приложения React должны правильно работать на устройствах с экранами разных размеров. Чтобы приложение функционировало должным образом, необходимо выполнить сравнительно простые модификации CSS, позволяющие настроить размер текста и организацию экранных элементов, а также более существенные модификации, в результате которых впечатления от перемещений по веб-сайту для пользователей мобильных и настольных устройств будут существенно различаться.

Приложение в нашем примере отображает список с именами и адресами людей. На рис. 2.1 показан вариант исполнения этого приложения на персональном компьютере.

Но такая организация элементов приложения не особенно хорошо подходит для работы на мобильных устройствах, экран которых может иметь достаточно места

только для отображения или списка имен людей, или их адресов, но не позволяет воспроизвести оба этих элемента одновременно.

Каким образом можно при помощи React предоставить пользователям мобильных и настольных устройств возможности работы с приложением, соответствующие их аппаратным средствам, не создавая при этом двух полностью разных приложений?

Рис. 2.1. Представление приложения, исполняющегося на настольном компьютере

РЕШЕНИЕ

Решение данной задачи состоит в создании реагирующих маршрутов. Реагирующий маршрут модифицируется соответственно размеру экрана пользователя. Для отображения информации об определенном человеке наше текущее приложение использует только один маршрут, а именно /people/:id.

При переходе по этому маршруту браузер отображает страницу, показанную на рис. 2.1. Здесь с левой стороны отображается список людей, а с правой — дополнительная информация о человеке, выбранном в этом списке.

Мы модифицируем наше приложение для обработки дополнительного маршрута по адресу /people. Затем мы сделаем каждый из этих маршрутов реагирующим, чтобы отображать информацию на устройствах разных типов соответствующим образом. Подробности этой модификации показаны в табл. 2.1.

Таблица 2.1. *Разные маршруты для разных типов устройств*

Маршрут	Мобильное устройство	Настольное устройство
`/people`	Отображается список людей	Выполняется переадресация по `people:someId`
`people:id`	Отображается подробная информация для выбранного `:id`	Отображается список людей и подробная информация для выбранного `:id`

Что нам требуется, чтобы реализовать этот подход? Прежде всего, если для приложения еще не установлена библиотека `react-router-dom`, нужно установить ее:

```
$ npm install react-router-dom
```

Библиотека `react-router-dom` позволяет координировать текущее местонахождение браузера с состоянием приложения. Далее необходимо установить библиотеку `react-media`, чтобы иметь возможность создавать компоненты React, реагирующие на изменения размеров экрана:

```
$ npm install react-media
```

Затем создадим реагирующий компонент `PeopleContainer` для управления маршрутами, которые мы хотим создать. На экранах небольшого размера этот компонент будет отображать или список людей, или подробную информацию для одного выбранного в этом списке человека. А на экранах большого размера он будет отображать объединенное представление списка людей слева и подробную информацию о выбранном человеке справа.

В компоненте `PeopleContainer` будет присутствовать компонент `Media` из библиотеки `react-media`, который функционирует подобно правилу `@media` CSS: позволяет создавать вывод для экранов с размерами из заданного диапазона. Он принимает в качестве параметра свойство `query`, позволяющее задавать набор экранных размеров. Для разделения экранов мобильных и настольных устройств мы определим только один размер экрана — малый (small), как показано в листинге 2.1.

Листинг 2.1. Определение размера экрана

```
<Media queries={{
        small: "(max-width: 700px)"
      }}>
   ...
</Media>
```

Компонент `Media` принимает в качестве параметра один компонент-потомок, ожидая, что это будет функция. Этой функции в качестве параметра передается объект `size`, указывающий размер текущего экрана. В нашем случае у объекта `size` будет атрибут `small`, который мы можем использовать, чтобы решить, какие другие компоненты отображать. В листинге 2.2 приведен полный код компонента `Media` с учетом этих дополнений.

Листинг 2.2. Полный код компонента `Media`

```
<Media queries={{
        small: "(max-width: 700px)"
      }}>
   {
      size => size.small ? [КОМПОНЕНТЫ ДЛЯ МАЛЫХ ЭКРАНОВ]
                         : [КОМПОНЕНТЫ ДЛЯ БОЛЬШИХ ЭКРАНОВ]
   }
</Media>
```

Прежде чем приступать к рассмотрению кода, возвращаемого для экранов разных размеров, полезно разобраться, каким образом компонент `PeopleContainer` будет вставляться в наше приложение. Для этого рассмотрим код основного компонента `App` (листинг 2.3).

Листинг 2.3. Код основного компонента App

```
import { BrowserRouter, Link, Route, Switch } from 'react-router-dom'
import PeopleContainer from './PeopleContainer'

function App() {
  return (
    <BrowserRouter>
      <Switch>
        <Route path="/people">
          <PeopleContainer />
        </Route>
        <Link to="/people">People</Link>
      </Switch>
    </BrowserRouter>
  )
}

export default App
```

Здесь компонент `BrowserRouter` из библиотеки `react-router-dom` связывает наш код и HTML5-историю API-интерфейса в браузере. Чтобы дать всем нашим маршрутам доступ к текущему адресу браузера, нам нужно поместить их в компонент `BrowserRouter`.

В компоненте `BrowserRouter` находится компонент `Switch`, который проверяет находящиеся в нем компоненты на наличие маршрута `Route`, соответствующего текущему местонахождению браузера. В данном случае имеется только один маршрут, задающий пути, начинающиеся с `/people`. В случае совпадения отображается компонент `PeopleContainer`. При отсутствии совпадений весь код до конца компонента `Switch` пропускается и отображается ссылка `Link` на путь `/people`. Поэтому, когда пользователь открывает домашнюю страницу приложения, он увидит только ссылку на страницу `People`.

Код будет считать совпадающими маршруты, начинающиеся с заданного пути, если только не задан точный атрибут. В таком случае маршрут будет отображаться только при условии совпадения полного пути.

Таким образом, если мы находимся внутри компонента `PeopleContainer`, значит, мы уже на маршруте, путь которого начинается с `/people/...` При исполнении приложения на устройстве с экраном малого размера нам нужно отобразить или список

людей, или подробную информацию для одного выбранного в этом списке человека, но не оба эти компонента. Данная задача выполняется с помощью компонента `Switch` (листинг 2.4).

Листинг 2.4. Код компонента `Switch` для выбора отображаемого компонента

```
<Media queries={{
        small: "(max-width: 700px)"
      }}>
   {
      size => size.small ? [КОМПОНЕНТЫ ДЛЯ МАЛЫХ ЭКРАНОВ]
         <Switch>
            <Route path='/people/:id'>
               <Person/>
            </Route>
            <PeopleList/>
         </Switch>
         : [КОМПОНЕНТЫ ДЛЯ БОЛЬШИХ ЭКРАНОВ]
   }
</Media>
```

При исполнении на устройстве с малым экраном компонент `Media` вызывает свою дочернюю функцию, передавая ей значение, соответствующее истинности выражения `size.small`. Код исполнит компонент `Switch`, который отрисует компонент `Person`, если текущий пусть содержит идентификатор `id`. В противном случае компонент `Switch` не найдет совпадения с данным маршрутом `Route` и отрисует компонент `PeopleList`.

Игнорируя то обстоятельство, что у нас еще не готов код для работы с большими экранами, если исполнить имеющийся код на мобильном устройстве и щелкнуть по ссылке `People` на домашней странице, то будет выполнен переход по пути people, в результате чего приложение отрисует компонент `PeopleList`. Данный компонент отображает набор ссылок на людей с путями в виде /people/id[1]. При выборе из этого списка какого-либо человека выполняется повторная отрисовка компонентов, но на этот раз компонент `PeopleContainer` отобразит подробности выбранного человека (рис. 2.2).

Пока что все идет по плану и можно двигаться дальше. А дальше нам нужно сделать так, чтобы приложение работало и на устройствах с большим экраном. Для этого нам необходимо генерировать реагирующие маршруты для компонента `PeopleContainer` для случаев неистинности выражения `size.small`. Если текущий маршрут представлен в виде /people/id, то можно одновременно отображать компоненты `PeopleList` и `Person` слева и справа соответственно (листинг 2.5).

---

[1] Здесь мы не приводим код для компонента `PeopleList`, но его можно загрузить на GitHub (**https://oreil.ly/tZzMD**).

Рис. 2.2. Исполнение приложения на мобильном устройстве: отображение списка людей (слева) со ссылками на подробную информацию о каждом человеке (справа)

Листинг 2.5. Одновременное отображение компонентов `PeopleList` и `Person`

```
<div style={{display: 'flex'}}>
  <PeopleList/>
  <Person/>
</div>
```

Но, к сожалению, этот код не учитывает случай, когда текущий путь просто /people. Нам нужно добавить другой компонент `Switch`, который будет или отображать подробную информацию для выбранного человека, или перенаправлять по пути /people/first-person-id для первого человека в списке (листинг 2.6).

Листинг 2.6. Компонент `Switch`

```
<div style={{display: 'flex'}}>
  <PeopleList/>
  <Switch>
    <Route path='/people/:id'>
      <Person/>
    </Route>
    <Redirect to={`/people/${people[0].id}`}/>
  </Switch>
</div>
```

Но компонент `Redirect` в действительности не выполняет собственно перенаправление браузера. Он просто обновляет текущий путь до значения `/people/first-person-`

id, в результате чего компонент PeopleContainer повторно отрисовывает страницу. Это похоже на вызов функции history.push() в JavaScript, с тем исключением, что к истории браузера не добавляется дополнительная страница. Если пользователь переходит до узла /people, то браузер просто обновляет свое местонахождение до /people/first-person-id.

Если теперь на настольном компьютере или планшете с большим экраном перейти к узлу /people, то на экране отобразится список людей, а справа от него — подробная информация о выбранном человеке (рис. 2.3).

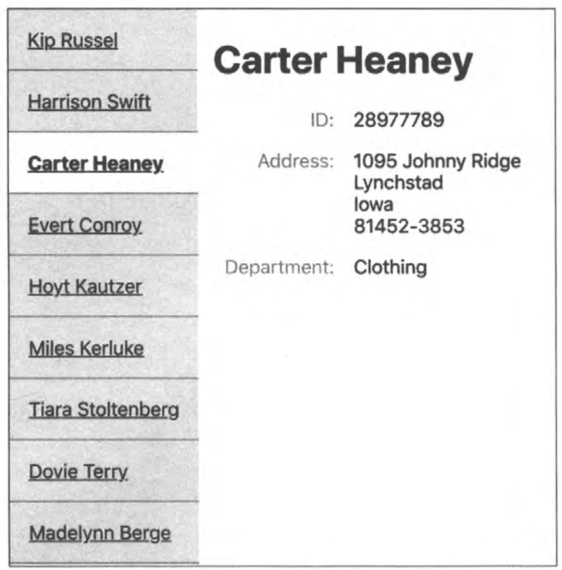


Рис. 2.3. Отображение на большом экране компонента по маршруту http://localhost:3000/people

Окончательная версия кода компонента PeopleContainer приведена в листинге 2.7.

Листинг 2.7. Окончательная версия кода компонента PeopleContainer

```
import Media from 'react-media'
import { Redirect, Route, Switch } from 'react-router-dom'
import Person from './Person'
import PeopleList from './PeopleList'
import people from './people'

const PeopleContainer = () => {
  return (
    <Media
      queries={{
        small: '(max-width: 700px)',
      }}
    >
```

```
        {(size) =>
          size.small ? (
            <Switch>
              <Route path="/people/:id">
                <Person />
              </Route>
              <PeopleList />
            </Switch>
          ) : (
            <div style={{ display: 'flex' }}>
              <PeopleList />
              <Switch>
                <Route path="/people/:id">
                  <Person />
                </Route>
                <Redirect to={`/people/${people[0].id}`} />
              </Switch>
            </div>
          )
        }
      </Media>
  )
}

export default PeopleContainer
```

Обсуждение

При первом знакомстве декларативная маршрутизация внутри компонентов может выглядеть непривычно. Для тех, кто раньше использовал только модель централизованной маршрутизации, декларативные маршруты на первых порах могут выглядеть неопрятными, поскольку они разбрасывают логику приложения по нескольким компонентам, вместо того, чтобы содержать ее в одном файле. Вместо создания компактных компонентов, которым не известно ничего о внешнем мире, мы создаем компоненты, обладающие всей информацией об используемых в приложении путях, что может отрицательно влиять на их портативность.

Но реагирующие маршруты демонстрируют настоящую мощь декларативной маршрутизации. Если вас беспокоит то, что компонентам известно слишком много о путях в приложении, возможен подход с сохранением строк путей в общедоступном файле. Таким образом у вас будут как компоненты, которые модифицируют свои действия в зависимости от текущего пути, так и централизованный набор конфигураций путей.

Исходный код данного рецепта можно загрузить на веб-сайте GitHub по адресу **https://oreil.ly/tZzMD**.

2.2. Размещение состояния в маршрутах

ЗАДАЧА

Часто бывает полезным управлять внутренним состоянием компонента, отображая его в маршруте. Например, в листинге 2.8 приведен код компонента React, который отображает две информационные вкладки: одна для маршрута /people, а другая для маршрута /offices.

Листинг 2.8. Код компонента React, отображающий две вкладки

```
import { useState } from 'react'
import People from './People'
import Offices from './Offices'

import './About.css'

const OldAbout = () => {
   const [tabId, setTabId] = useState('people')

   return (
      <div className="About">
         <div className="About-tabs">
            <div
               onClick={() => setTabId('people')}
               className={
                  tabId === 'people' ? 'About-tab active' : 'About-tab'
               }
            >
               People
            </div>
            <div
               onClick={() => setTabId('offices')}
               className={
                  tabId === 'offices' ? 'About-tab active' : 'About-tab'
               }
            >
               Offices
            </div>
         </div>
         {tabId === 'people' && <People />}
         {tabId === 'offices' && <Offices />}
      </div>
   )
}

export default OldAbout
```

Когда пользователь щелкнет по одной из вкладок, обновляется переменная `tabid` и отображается соответствующий компонент `People` или `Office` (рис. 2.4).

Но с этим кодом есть одна проблема. В частности, компонент в целом работает, но если выбрать вкладку **Offices**, а затем обновить страницу, компонент сбрасывается снова к вкладке **People**. В результате для страницы нельзя создать закладку на вкладке **Offices**. Более того, создать ссылку, ведущую непосредственно на вкладку **Offices,** не возможно нигде в приложении. Также снижается вероятность определения работы вкладок в качестве гиперссылок оборудованием для обеспечения специальных возможностей, поскольку они не отрисовываются таким образом.

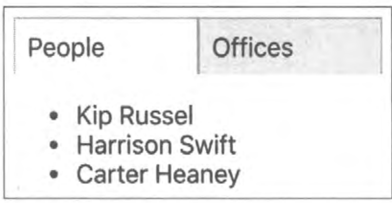


Рис. 2.4. По умолчанию компонент `OldAbout` отображает подробную информацию о людях

РЕШЕНИЕ

Для решения данной проблемы мы переместим состояние `tabid` из компонента в текущее местонахождение браузера. Таким образом, вместо отрисовки компонента в узле /about с последующим использованием событий `onClick` для изменения внутреннего состояния, мы будем использовать маршруты к /about/people и /about/offices, отображающие соответствующие вкладки. Выбор вкладок, таким образом, выстоит обновление страницы браузером. Также мы сможем создать ссылку или закладку страницы для данной вкладки.

Наконец, мы сделаем вкладки настоящими гиперссылками, которые будут определяться как таковые при перемещении по странице посредством как клавиатуры, так и экранного считывателя.

Чтобы реализовать все это, нам нужно установить только одну библиотеку: `react-router-dom`:

```
$ npm install react-router-dom
```

Библиотека `react-router-dom` позволит нам синхронизировать текущий URL-адрес браузера с отображаемыми на экране компонентами.

Она уже используется в нашем существующем приложении для отображения компонента `OldAbout` по пути /oldabout, как показано во фрагменте кода из файла App.js (листинг 2.9).

Листинг 2.9. Код из файла App.js для отображения компонента `OldAbout`

```
<Switch>
  <Route path="/oldabout">
```

```
        <OldAbout/>
    </Route>
    <p>Choose an option</p>
</Switch>
```

Полный код из этого файла можно загрузить из репозитория GitHub по адресу **https://oreil.ly/WmZ18**.

Мы создадим новую версию компонента `OldAbout`, называющуюся `About`, и установим его по своему собственному маршруту (листинг 2.10).

Листинг 2.10. Установка компонента `About` по своему маршруту

```
<Switch>
    <Route path="/oldabout">
        <OldAbout/>
    </Route>
    <Route path="/about/:tabId?">
        <About/>
    </Route>
    <p>Choose an option</p>
</Switch>
```

Таким образом мы сможем использовать обе версии кода в нашем примере приложения.

Новая версия компонента будет выглядеть идентично старой. Мы извлечем идентификатор `tabid` из компонента и переместим его по текущему пути.

В результате присвоения компоненту `Route` пути /about/:tabId? наш компонент будет устанавливаться всеми тремя маршрутами: /about, /about/offices и /about/people. Символ знака вопроса (?) означает, что параметр `tabid` не является обязательным.

Первая часть задачи выполнена: мы поместили состояние компонента в путь, который отображает его. Теперь нам нужно обновить компонент, чтобы взаимодействовать с маршрутом, а не с переменной внутреннего состояния.

Обе вкладки компонента `OldAbout` были подписаны на оповещения от `onClick` (листинг 2.11).

Листинг 2.11. Подписка вкладок People и Offices на оповещения от `onClick`

```
<div onClick={() => setTabId("people")}
    className={tabId === "people" ? "About-tab active" : "About-tab"}
>
    People
</div>
<div onClick={() => setTabId("offices")}
    className={tabId === "offices" ? "About-tab active" : "About-tab"}
>
```

```
  Offices
</div>
```

Мы преобразуем эти подписки в ссылочные компоненты `Link`, ведущие к /about/people и /about/offices. Более точно мы преобразуем их в компоненты `NavLink`. Компонент `NavLink` похож на ссылку, но обладает возможностью задавать дополнительное название класса, если он ссылается на текущее местонахождение. В результате в исходном коде можно обойтись без логики `className` (листинг 2.12).

Листинг 2.12. Использовании ссылочного компонента NavLink

```
<NavLink to="/about/people"
         className="About-tab"
         activeClassName="active">
      People
</NavLink>
<NavLink to="/about/offices"
         className="About-tab"
         activeClassName="active">
      Offices
</NavLink>
```

Значение переменной `tabId` больше не задается. Вместо этого мы переходим на новую страницу с новым значением `tabId` в пути.

Но каким образом мы считываем это значение `tabid`? В коде компонента `OldAbout` содержимое текущей вкладки отображается следующим кодом:

```
{tabId === "people" && <People/>}
{tabId === "offices" && <Offices/>}
```

Этот код можно заменить компонентом `Switch` и парой компонентов `Route`, как показано в листинге 2.13.

Листинг 2.13. Новый код для отображения содержимого текущей вкладки

```
<Switch>
   <Route path='/about/people'>
      <People/>
   </Route>
   <Route path='/about/offices'>
      <Offices/>
   </Route>
</Switch>
```

Все почти готово. Нам осталось только решить, что делать для пути `/about`, который не содержит `tabId`.

В старом компоненте `OldAbout` переменной `tabId` присваивается значение по умолчанию при первом создании состояния:

```
const [tabId, setTabId] = useState("people")
```

Этот же эффект можно получить, добавив в конец кода компонента `Switch` команду `Redirect`. Компонент `Switch` будет по порядку обрабатывать свои дочерние компоненты, пока не найдет совпадающий маршрут `Route`. Если для текущего пути не будет обнаружено совпадающего маршрута, исполнение перейдет к команде `Redirect`, которая изменит адрес на /about/people. В результате выполняется повторная отрисовка компонента `About` и по умолчанию выбирается вкладка `People` (листинг 2.14).

Листинг 2.14. Код для отображения содержимого текущей вкладки с добавленной командой Redirect

```
<Switch>
  <Route path='/about/people'>
    <People/>
  </Route>
  <Route path='/about/offices'>
    <Offices/>
  </Route>
  <Redirect to='/about/people'/>
</Switch>
```

Исполнение команды `Redirect` можно сделать зависимым от значения текущего пути, снабдив ее атрибутом `from`. В таком случае атрибуту `from` можно присвоить значение `/about`, чтобы по маршруту /about/people перенаправлялись только маршруты, совпадающие с /about.

Полный код нового компонента `About` приведен в листинге 2.15.

Листинг 2.15. Полный код нового компонента About

```
import { NavLink, Redirect, Route, Switch } from 'react-router-dom'
import './About.css'
import People from './People'
import Offices from './Offices'

const About = () => (
  <div className="About">
    <div className="About-tabs">
      <NavLink
        to="/about/people"
        className="About-tab"
        activeClassName="active"
      >
```

```
            People
        </NavLink>
        <NavLink
            to="/about/offices"
            className="About-tab"
            activeClassName="active"
        >
            Offices
        </NavLink>
    </div>
    <Switch>
        <Route path="/about/people">
            <People />
        </Route>
        <Route path="/about/offices">
            <Offices />
        </Route>
        <Redirect to="/about/people" />
    </Switch>
  </div>
)

export default About
```

Нам больше не нужна внутренняя переменная `tabID`, и теперь мы имеем чисто декларативный компонент (рис. 2.5).

Рис. 2.5. Переход по адресу http://localhost/about/offices посредством нового компонента

Обсуждение

Перенос состояния из компонентов в строку адреса может упростить код, но это всего лишь побочный эффект, который по счастливой случайности оказался полез-

ным. Настоящая польза от этого состоит в том, что наше приложение начинает вести себя в меньшей степени как приложение и в большей как веб-сайт. Мы можем делать закладки страниц, а кнопки **Назад** и **Вперед** браузера работают правильно. Перенос управления в маршруты — это не какое-то абстрактное решение для целей разработки, а способ сделать поведение приложения более ожидаемым для пользователей.

Исходный код данного рецепта можно загрузить на веб-сайте GitHub по адресу **https://oreil.ly/myAGj**.

2.3. Модульное тестирование посредством MemoryRouter

ЗАДАЧА

В приложениях React маршруты используются для того, чтобы реализовать возможности браузера в большем объеме. Это позволяет делать закладки страниц, создавать ссылки глубоко в приложение, а также перемещаться вперед и назад по истории просмотра страниц.

Но, начиная применять маршруты, мы делаем компонент зависимым от иных факторов, чем он сам, а именно от URL-адреса браузера. Это может показаться не такой и большой проблемой, но тем не менее имеет свои последствия.

Предположим, что мы хотим выполнить модульное тестирование компонента, осведомленного о своем текущем маршруте. Для примера осуществим модульное тестирование компонента `About` из *раздела 2.2* (листинг 2.16)[2].

Листинг 2.16. Код модульного тестирования для компонента About

```
describe('About component', () => {
  it('should show people', () => {
    render(<About />)
    expect(screen.getByText('Kip Russel')).toBeInTheDocument()
  })
})
```

Модульное тестирование отрисовывает компонент, а затем ищет в выводе имя "Kip Russel". Но в результате выполнения этого теста выдается следующее сообщение об ошибке:

```
console.error node_modules/jsdom/lib/jsdom/virtual-console.js:29
Error: Uncaught [Error: Invariant failed: You should not use <NavLink>
outside a <Router>]
```

[2] В этом примере используется библиотека React Testing Library.

Ошибка была вызвана тем, что компонент NavLink не смог найти компонент Router, расположенный выше в дереве компонентов. Это означает, что прежде чем тестировать компонент, его необходимо обернуть в компонент Router.

Кроме того, нам может потребоваться создать модульный тест, который проверяет, что компонент About работает должным образом при его установке на каком-либо конкретном маршруте. Но даже если мы предоставим оболочку в виде компонента Router, как нам эмулировать конкретный маршрут?

Данная проблема присуща не только модульному тестированию. При использовании библиотечного средства типа Storybook[3] полезно продемонстрировать, как компонент будет выглядеть после его установки в данном пути.

Нам нужно что-то наподобие настоящего браузерного маршрутизатора, но с возможностью управлять его поведением.

РЕШЕНИЕ

Библиотека react-router-dom содержит как раз такой маршрутизатор MemoryRouter. Для внешнего мира данный маршрутизатор выглядит точно так же, как маршрутизатор BrowserRouter. Разница между этими двумя маршрутизаторами состоит в том, что маршрутизатор BrowserRouter является интерфейсом для нижележащего API истории браузера, а маршрутизатор MemoryRouter такой зависимости не имеет. Он может отслеживать текущее местонахождение и перемещаться вперед и назад по истории, но делает это с помощью простых структур памяти.

Взглянем еще раз на наш модульный тест, выдающий ошибку. Но на этот раз вместо простой отрисовки компонента About обернем его в компонент MemoryRouter (листинг 2.17).

Листинг 2.17. Компонент About в обертке компонента MemoryRouter

```
describe('About component', () => {
  it('should show people', () => {
    render(
      <MemoryRouter>
        <About />
      </MemoryRouter>
    )

    expect(screen.getByText('Kip Russel')).toBeInTheDocument()
  })
})
```

Теперь, поскольку компонент MemoryRouter вставляет в контекст эмулированную версию API, тест работает должным образом, т. к. это делает ее доступной для всех

[3] См. *раздел 1.9*.

его дочерних компонентов. Теперь благодаря доступности истории компонент About может отрисовывать ссылку Link или маршрут Route.

Но компонент MemoryRouter обладает дополнительным преимуществом. Поскольку он эмулирует API истории браузера, ему можно присвоить полностью выдуманную историю при помощи свойства initialEntries. Этому свойству нужно задать значение массива элементов истории. Массив с одним значением рассматривается как текущий URL-адрес. Таким образом можно создавать модульные тесты для проверки работы компонента, установленного на определенном маршруте (листинг 2.18).

Листинг 2.18. Модульный тест компонента, установленного на определенном маршруте

```
describe('About component', () => {
  it('should show offices if in route', () => {
    render(
      <MemoryRouter initialEntries={[{ pathname: '/about/offices' }]}>
        <About />
      </MemoryRouter>
    )

     expect(screen.getByText('South Dakota')).toBeInTheDocument()
  })
})
```

Можно было бы использовать компонент BrowserRouter средства Storybook, как в настоящем браузере, но компонент MemoryRouter также позволяет эмулировать текущий URL, как это делается в рассказе ToAboutOffices средства Storybook (рис. 2.6).

Обсуждение

Маршрутизаторы позволяют разделить подробности конечного пункта перехода и то, как этот переход реализуется. Из данного раздела очевидно одно из преимуществ такого подхода: мы можем эмулировать URL браузера, чтобы исследовать работу компонента на разных маршрутах. Такое разделение позволяет изменять способ перехода приложения по ссылкам без угрозы сбоя. В случае преобразования одностраничного приложения в приложение с отрисовкой на сервере, компонент BrowserRouter заменяется компонентом StaticRouter. Ссылки, которые раньше использовались для вызовов API истории браузера, станут нативными гиперссылками, вызывающими загрузку браузером нативных страниц. Маршрутизаторы предоставляют хороший пример преимущества метода отделения политики (что нужно сделать) от механизмов (как это сделать).

Исходный код данного рецепта можно загрузить на веб-сайте GitHub по адресу **https://oreil.ly/1NW8e**.

Рис. 2.6. Компонент `MemoryRouter` позволяет эмулировать маршрут /about/offices

2.4. Подтверждение ухода со страницы посредством компонента Prompt

ЗАДАЧА

Иногда будет полезным уточнить у пользователя, находящегося в процессе выполнения какой-либо задачи, например редактирования текста, действительно ли он хочет уйти с данной страницы. Пример, как может выглядеть представление такого запроса, показан на рис. 2.7.

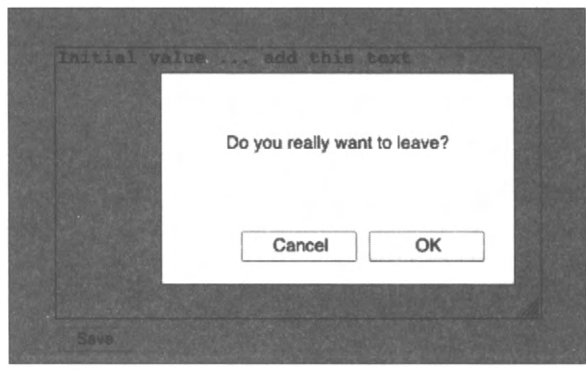


Рис. 2.7. Запрос подтверждения ухода с текущей страницы

Эта на первый взгляд простая задача может оказаться сложной, поскольку требует обнаружения нажатия пользователем кнопки **Назад**, а затем реализации каким-либо способом перехвата перехода назад в истории браузера и, возможно, отмены этого перехода. Что если такой возможностью нужно оснастить несколько страниц приложения? Существует ли простой способ создать эту возможность так, чтобы ее можно было использовать в любом нуждающемся в ней компоненте?

РЕШЕНИЕ

Данную задачу можно решить с помощью компонента `Prompt` библиотеки `react-router-dom`, который запрашивает у пользователей подтверждение, действительно ли они хотят уйти с данной страницы.

Единственная составляющая, необходимая для реализации данного решения, — это сама библиотека `react-router-dom`. Установим ее:

```
npm install react-router-dom
```

Предположим, что по маршруту /important у нас находится компонент `Important`, который позволяет пользователю редактировать текст (листинг 2.19).

Листинг 2.19. Компонент `Important` для редактирования текста

```
import React, { useEffect, useState } from 'react'

const Important = () => {
   const initialValue = 'Initial value'

   const [data, setData] = useState(initialValue)
   const [dirty, setDirty] = useState(false)

   useEffect(() => {
      if (data !== initialValue) {
        setDirty(true)
      }
   }, [data, initialValue])

   return (
      <div className="Important">
         <textarea
            onChange={(evt) => setData(evt.target.value)}
            cols={40}
            rows={12}
         >
            {data}
         </textarea>
         <br />
         <button onClick={() => setDirty(false)} disabled={!dirty}>
```

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

export default Important
```

Компонент `Important` уже самостоятельно отслеживает, отличается ли текст в поле `textarea` от исходного значения. Если да, то переменной `dirty` присваивается значение `true`. Но как нам спросить у пользователя, нажавшего кнопку **Назад** при установленной переменной `dirty`, действительно ли он хочет уйти со страницы?

Добавив в код компонент `Prompt`, как показано в листинге 2.20.

Листинг 2.20. Добавление в код компонента `Prompt`

```
return (
   <div className="Important">
      <textarea
         onChange={(evt) => setData(evt.target.value)}
         cols={40}
         rows={12}
      >
         {data}
      </textarea>
      <br />
      <button onClick={() => setDirty(false)} disabled={!dirty}>
         Save
      </button>
      <Prompt
         when={dirty}
         message={() => 'Do you really want to leave?'}
      />
   </div>
)
```

При нажатии пользователем кнопки **Назад** в процессе редактирования текста компонент `Prompt` выводит на экран диалоговое окно с запросом подтвердить уход со страницы (рис. 2.8).

Как видим, добавить запрос на подтверждение несложно, но при этом по умолчанию выводится простое диалоговое окно функции `prompt` JavaScript. Было бы неплохо иметь выбор, каким именно образом запрашивать подтверждение ухода.

Реализовать такую возможность также не представляет большой сложности. Чтобы продемонстрировать это, добавим в приложение библиотеку компонентов Material-UI:

```
$ npm install '@material-ui/core'
```

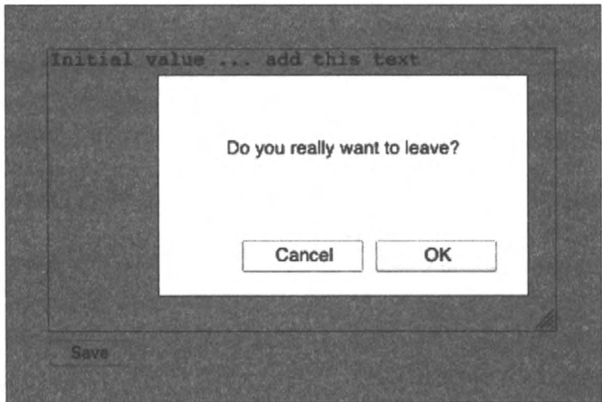


Рис. 2.8. Диалоговое окно с запросом подтвердить уход со страницы

Эта библиотека реализует в React стандарт Material Design компании Google. Мы используем ее в качестве примера, насколько легко заменить стандартное диалоговое окно приглашения чем-то более индивидуальным.

В действительности компонент `Prompt` не отрисовывает никаких элементов пользовательского интерфейса. Он просто подает текущему маршрутизатору запрос на отображение запроса на подтверждение. По умолчанию маршрутизатор `BrowserRouter` отображает стандартное диалоговое окно приглашения JavaScript. Это окно можно заменить на специализированное пользовательское окно.

При добавлении маршрутизатора `BrowserRouter` в дерево компонентов ему можно передать свойство, называющееся `getUserConfirmation`, как показано в листинге 2.21.

Листинг 2.21. Добавление в дерево компонентов свойства `getUserConfirmation`

```
<div className="App">
  <BrowserRouter
    getUserConfirmation={(message, callback) => {
      // Custom code goes here
    }}
  >
    <Switch>
      <Route path='/important'>
        <Important/>
      </Route>
    </Switch>
  </BrowserRouter>
</div>
```

Свойство `getUserConfirmation` представляет собой функцию, которая принимает два входных параметра: сообщение, которое нужно отобразить, и функцию обратного вызова.

Когда пользователь нажимает кнопку **Назад**, компонент `Prompt` исполняет функцию `getUserConfirmation`, а затем ожидает перехода к функции обратного вызова со значением `true` или `false`.

Функция обратного вызова возвращает ответ пользователя асинхронно. Компонент `Prompt` будет ожидать до тех пор, пока мы не спросим пользователя, что он хочет делать. Вот таким образом мы можем создать индивидуализированный интерфейс.

Создадим средствами библиотеки Material-UI пользовательское диалоговое окно `Alert`, которое будем выводить вместо стандартного диалогового окна приглашения JavaScript (листинг 2.22).

Листинг 2.22. Пользовательское диалоговое окно на основе библиотеки Material-UI

```
import Button from '@material-ui/core/Button'
import Dialog from '@material-ui/core/Dialog'
import DialogActions from '@material-ui/core/DialogActions'
import DialogContent from '@material-ui/core/DialogContent'
import DialogContentText from '@material-ui/core/DialogContentText'
import DialogTitle from '@material-ui/core/DialogTitle'

const Alert = ({ open, title, message, onOK, onCancel }) => {
   return (
      <Dialog
         open={open}
         onClose={onCancel}
         aria-labelledby="alert-dialog-title"
         aria-describedby="alert-dialog-description"
      >
         <DialogTitle id="alert-dialog-title">{title}</DialogTitle>
         <DialogContent>
            <DialogContentText id="alert-dialog-description">
               {message}
            </DialogContentText>
         </DialogContent>
         <DialogActions>
            <Button onClick={onCancel} color="primary">
               Cancel
            </Button>
            <Button onClick={onOK} color="primary" autoFocus>
               OK
            </Button>
         </DialogActions>
      </Dialog>
   )
}

export default Alert
```

Конечно же, какой-либо особой причины, по которой мы должны выводить именно диалоговое окно, у нас нет. Можно было бы отобразить обратный счетчик или сообщение SnackBar[4] или автоматически сохранить все выполненные пользователем правки. Но мы решили выводить индивидуализированное диалоговое окно **Alert**.

Чтобы использовать компонент `Alert` в нашем интерфейсе, первым делом нам нужно создать свою функцию `getUserConfirmation`. Мы сохраним сообщение и функцию обратного вызова, а затем зададим значение булевой переменной, сообщающее, что мы хотим открыть диалоговое окно **Alert**. Соответствующий код приведен в листинге 2.23.

Листинг 2.23. Код для реализации функции `getUserConfirmation`

```
const [confirmOpen, setConfirmOpen] = useState(false)
const [confirmMessage, setConfirmMessage] = useState()
const [confirmCallback, setConfirmCallback] = useState()

return (
  <div className="App">
    <BrowserRouter
      getUserConfirmation={(message, callback) => {
        setConfirmMessage(message)
        // Use this setter form because callback is a function
        // Используем данную установочную форму, поскольку обратный вызов является
                                                                          функцией
        setConfirmCallback(() => callback)
        setConfirmOpen(true)
      }}
    >
  .....
```

Следует заметить, что при сохранении функции обратного вызова вместо простого оператора `setConfirmCallback(callback)` применяется оператор `setConfirmCallback(() => callback)`. Это объясняется тем, что возвращаемые хуком `useState` сеттеры (setter — установщик) не сохраняют передаваемые им функции, а исполняют их.

Затем мы можем задать значения `confirmMessage`, `confirmCallback` и `confirmOpen` для отрисовки компонента `Alert` в интерфейсе. В листинге 2.24 приведен полный код решения из файла App.js.

Листинг 2.24. Полный код решения

```
import { useState } from 'react'
import './App.css'
```

[4] SnackBar — виджет библиотеки Material Design, который используется для отображения сообщений в нижней части приложения.

```
import { BrowserRouter, Link, Route, Switch } from 'react-router-dom'
import Important from './Important'
import Alert from './Alert'

function App() {
  const [confirmOpen, setConfirmOpen] = useState(false)
  const [confirmMessage, setConfirmMessage] = useState()
  const [confirmCallback, setConfirmCallback] = useState()

  return (
    <div className="App">
      <BrowserRouter
        getUserConfirmation={(message, callback) => {
          setConfirmMessage(message)
          // Use this setter form because callback is a function
          setConfirmCallback(() => callback)
          setConfirmOpen(true)
        }}
      >
        <Alert
          open={confirmOpen}
          title="Leave page?"
          message={confirmMessage}
          onOK={() => {
            confirmCallback(true)
            setConfirmOpen(false)
          }}
          onCancel={() => {
            confirmCallback(false)
            setConfirmOpen(false)
          }}
        />
        <Switch>
          <Route path="/important">
            <Important />
          </Route>
          <div>
            <h1>Home page</h1>
            <Link to="/important">Go to important page</Link>
          </div>
        </Switch>
      </BrowserRouter>
    </div>
  )
}

export default App
```

Теперь, когда пользователь прекращает редактирование текста, возвращаясь на предыдущую страницу, отображается индивидуализированное диалоговое окно с запросом подтвердить это действие (рис. 2.9).

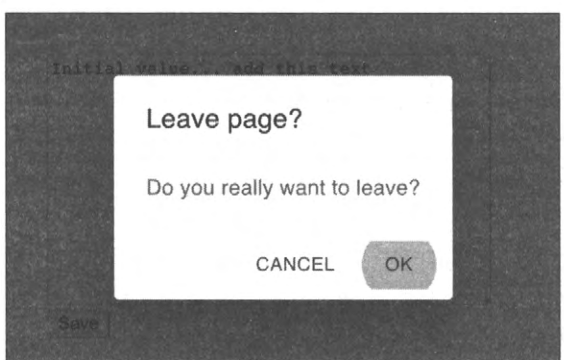


Рис. 2.9. Индивидуализированное диалоговое окно **Alert**

Обсуждение

В этом разделе мы реализовали собственную версию диалогового окна **Promt**, используя библиотеку компонентов. Но это не означает, что мы ограничены простой заменой одного диалогового окна другим. Нет абсолютно никакой причины, почему при попытке пользователя уйти со страницы мы не могли бы делать что-то другое, например сохранить выполняемую им работу, чтобы он мог вернуться к ней в дальнейшем. Асинхронная природа функции `getUserConfirmation` делает такую гибкость возможной. Это еще один пример, как библиотека `react-router-dom` абстрагирует сквозную функциональность, позволяя разработчику не заниматься подробностями ее реализации.

Исходный код данного рецепта можно загрузить на веб-сайте GitHub по адресу **https://oreil.ly/1FyoE**.

2.5. Создание переходов посредством библиотеки React Transition Group

ЗАДАЧА

В нативных и десктопных приложениях часто встречается анимация для визуального соединения разных элементов. Например, щелчок по элементу списка вызывает его разворачивание и подробное отображение всего содержимого. А смахивание влево или вправо можно использовать для обозначения, принимает ли пользователь предлагаемую опцию или нет.

Таким образом, анимационные эффекты часто применяются для обозначения изменений URL-адреса. Они укрупняют элементы, позволяя рассмотреть их подробно-

сти, выполняют переход от одного элемента списка к другому. Также можно отражать изменения URL-адреса соответствующей анимацией.

Но как создать анимацию при переходе от одного адреса к другому?

РЕШЕНИЕ

Для решения данной задачи нам потребуются библиотеки `react-router-dom` и `react-transition-group`:

```
$ npm install react-router-dom
$ npm install react-transition-group
```

Будем анимировать компонент `About`, с которым мы работали в предыдущих разделах[5]. Компонент `About` имеет две вкладки — **People** и **Offices**, которые отображаются для маршрутов /about/people и /about/offices соответственно.

При щелчке по любой из этих вкладок содержимое старой вкладки будет удалено с эффектом постепенного исчезновения (fade-out) и заменено новым содержимым с постепенным появлением (fade-in). Хотя в данном случае мы используем эффекты постепенного исчезновения и появления, с таким же успехом мы могли бы реализовать и более сложную анимацию, например эффект скольжения влево или вправо для удаления и помещения содержимого вкладок на экран[6]. Но простая анимация на основе эффектов постепенного исчезновения и появления позволит более ясно продемонстрировать сам принцип работы.

В компоненте `About` содержимое вкладок отрисовывается компонентами `People` и `Offices` в соответствующих маршрутах (листинг 2.25).

Листинг 2.25. Код для отрисовки вкладок People и Offices

```
import { NavLink, Redirect, Route, Switch } from 'react-router-dom'
import './About.css'
import People from './People'
import Offices from './Offices'
const About = () => (
  <div className="About">
    <div className="About-tabs">
      <NavLink
        to="/about/people"
        className="About-tab"
        activeClassName="active"
      >
        People
      </NavLink>
```

[5] См. *разделы 2.2* и *2.3*.

[6] Это часто встречается в компонентах с вкладками сторонних разработчиков. Подобная анимация укрепляет в уме пользователей впечатление движения слева направо по вкладкам.

```
        <NavLink
          to="/about/offices"
          className="About-tab"
          activeClassName="active"
        >
          Offices
        </NavLink>
      </div>
      <Switch>
        <Route path="/about/people">
          <People />
        </Route>
        <Route path="/about/offices">
          <Offices />
        </Route>
        <Redirect to="/about/people" />
      </Switch>
    </div>
)
export default About
```

Теперь нам нужно анимировать компоненты, находящиеся внутри компонента `Switch`. Для этого нам потребуются следующие две вещи:

- способ для отслеживания изменения местонахождения (URL-адреса);
- способ для анимации содержимого вкладки при таком изменении.

Узнать об изменении местонахождения браузера можно, получая его текущее местонахождение из хука `useLocation` библиотеки `react-router-dom`:

```
const location = useLocation()
```

Теперь нам осталось решить более сложную задачу: собственно анимацию. Далее приводится довольно сложная последовательность событий, но будет полезным не спешить и полностью разобраться с тем, что происходит.

При переходе от одного компонента к другому с эффектом анимации нам нужно удерживать на странице оба компонента. В то время как компонент `Office` постепенно исчезает, компонент `People` постепенно появляется[7]. Этого можно добиться, удерживая оба компонента в группе перехода. Группа перехода — это набор компонентов, некоторые из которых исчезают, а другие появляются.

Такую группу можно создать, заключив нашу анимацию в обертку из компонента `TransitionGroup`. Нам также потребуется компонент `CSSTransition`, чтобы координировать подробности анимации CSS.

Таким образом, в обновленном коде компонент `Switch` заключен в двойную оболочку компонентов `TransitionGroup` и `CSSTransition` (листинг 2.26).

---

[7] Для размещения обоих компонентов в одном и том же месте в процессе исчезновения и появления в коде используется относительное позиционирование.

Листинг 2.26. Обновленный код с оболочками `TransitionGroup` и `CSSTransition`

```
import {
   NavLink,
   Redirect,
   Route,
   Switch,
   useLocation,
} from 'react-router-dom'
import People from './People'
import Offices from './Offices'
import {
   CSSTransition,
   TransitionGroup,
} from 'react-transition-group'
import './About.css'
import './fade.css'
const About = () => {
   const location = useLocation()
   return (
      <div className="About">
         <div className="About-tabs">
            <NavLink
               to="/about/people"
               className="About-tab"
               activeClassName="active"
            >
               People
            </NavLink>
            <NavLink
               to="/about/offices"
               className="About-tab"
               activeClassName="active"
            >
               Offices
            </NavLink>
         </div>
         <TransitionGroup className="About-tabContent">
            <CSSTransition
               key={location.key}
               classNames="fade"
               timeout={500}
            >
               <Switch location={location}>
                  <Route path="/about/people">
                     <People />
                  </Route>
```

```
                <Route path="/about/offices">
                    <Offices />
                </Route>
                <Redirect to="/about/people" />
            </Switch>
        </CSSTransition>
    </TransitionGroup>
  </div>
 )
}
export default About
```

Обратите внимание на то, что ключу группы `CSSTransition` передается значение `location.key`, а компоненту `Switch` передается значение `location`. Значение `location.key` — это хешированное значение текущего URL-адреса. Передача этого значения группе перехода будет удерживать компонент `CSSTransition` в виртуальной модели DOM до завершения анимации. Выбор пользователем одной из вкладок изменяет URL-адрес, в результате чего выполняется обновление компонента `About`. Компонент `TransitionGroup` будет удерживать существующий компонент `CSSTransiton` в дереве компонентов до наступления тайм-аута, т. е. в течение 500 миллисекунд. Но теперь у него также будет и второй компонент `CSSTransition`.

Каждый из этих компонентов `CSSTransition` будет сохранять активными свои дочерние компоненты (рис. 2.10).

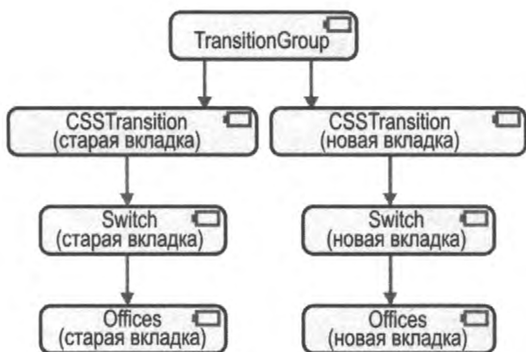


Рис. 2.10. Компонент `TransitionGroup` удерживает старый и новый компоненты `CSSTransition` активными в виртуальной модели DOM

Нам нужно передать значения URL-адресов компонентам `Switch`: одному для старой вкладки, а другому — для новой, которые нам нужны для того, чтобы продолжать отрисовывать свои маршруты.

Теперь можно приступать к реализации собственно анимации. Компонент `CSSTransition` принимает в качестве параметра свойство `classNames`, которому было присвоено значение `fade`. Обратите внимание на множественное число в названии свойства `classNames`. Это нужно для того, чтобы отличить его от стандартного атрибута `className`.

Компонент `CSSTransition` использует свойство `classNames` для создания четырех разных классов:

- `fade-enter`
- `fade-enter-active`
- `fade-exit`
- `fade-exit-active`

Класс `fade-enter` включает компоненты, которые будут выводиться на экран в скором времени. Класс `fade-enter-active` содержит компоненты, которые отображаются в настоящее время. А классы `fade-exit` и `fade-exit-active` содержат компоненты, которые начинают исчезать или находятся в процессе анимации исчезновения соответственно.

Эти названия классов будут добавлены в компоненте `CSSTransition` их непосредственным потомкам. При выполнении анимации из вкладки **Offices** на вкладку **People** старый компонент `CSSTransition` добавит класс `fade-active` в код HTML вкладки **People**, а в код HTML вкладки **Offices** добавит класс `fade-exit-active`.

Теперь осталось только определить сами процессы анимации CSS (листинг 2.27).

Листинг 2.27. Определение анимаций CSS

```
.fade-enter {
  opacity: 0;
}
.fade-enter-active {
  opacity: 1;
  transition: opacity 250ms ease-in;
}
.fade-exit {
  opacity: 1;
}
.fade-exit-active {
  opacity: 0;
  transition: opacity 250ms ease-in;
}
```

В классах `fade-enter` коэффициент непрозрачности компонентов плавно изменяется от 0 до 1 за счет переходов CSS. А в классах `fade-exit` коэффициент непрозрачности изменяется обратно от 1 до 0 аналогичным образом. Обычно целесообразно хранить определения анимационных классов в отдельном файле CSS. Это позволит использовать их для других анимаций.

На этом задача анимации решена. При выборе пользователем одной из вкладок содержимое старой вкладки будет постепенно исчезать из вида, а ему на смену также постепенно будет появляться содержимое новой вкладки (рис. 2.11).

Рис. 2.11. Содержимое вкладки **Offices** постепенно заменяется содержимым вкладки **People**

Обсуждение

При некачественном исполнении анимации могут быть довольно раздражительными. Для применения каждой анимации должно быть какое-то основание. Если вы выбираете анимацию просто потому, что она вам лично нравится, почти наверняка пользователям она нравиться не будет. В общем, прежде чем реализовать анимацию, лучше всего выяснить некоторые аспекты:

- Сделает ли анимация взаимосвязь между двумя маршрутами более ясной? Применяется ли анимация для увеличения изображения, чтобы можно было увидеть его подробности, или для продольного перемещения, чтобы можно было увидеть какой-либо элемент, имеющий отношение к текущему элементу?
- Сколько должна длиться анимация? Больше чем полсекунды будет, скорее всего, слишком долго.
- Как анимация отразится на производительности? Переходы CSS обычно оказывают минимальный эффект на производительность приложения, если браузер передает соответствующую работу графическому процессору. Но что будет в случае старого браузера на мобильном устройстве?

Исходный код данного рецепта можно загрузить на веб-сайте GitHub по адресу **https://oreil.ly/UCu75**.

2.6. Создание защищенных маршрутов

ЗАДАЧА

Во многих приложениях возникает необходимость предотвращения доступа к определенным маршрутам до тех пор, пока пользователь не выполнит вход. Но как можно защитить одни маршруты, оставив открытым доступ к другим? Можно ли отделить механизмы безопасности от элементов пользовательского интерфейса для входа в систему и выхода из нее? Как это реализовать, не генерируя большой объем кода?

РЕШЕНИЕ

Рассмотрим один из способов реализации защиты маршрутов в приложении React. Данное приложение содержит домашнюю страницу (маршрут /), общедоступную страницу со свободным доступом (маршрут /public), а также две закрытые страницы (маршруты /private1 и /private2), которые нужно защитить. Код приложения приведен в листинге 2.28.

Листинг 2.28. Код приложения с маршрутами, требующими защиты

```
import React from 'react'
import './App.css'
import { BrowserRouter, Route, Switch } from 'react-router-dom'
import Public from './Public'
import Private1 from './Private1'
import Private2 from './Private2'
import Home from './Home'
function App() {
   return (
      <div className="App">
         <BrowserRouter>
            <Switch>
               <Route exact path="/">
                  <Home />
               </Route>
               <Route path="/private1">
                  <Private1 />
               </Route>
               <Route path="/private2">
                  <Private2 />
               </Route>
               <Route exact path="/public">
                  <Public />
               </Route>
            </Switch>
            </BrowserRouter>
         </div>
      )
   }
export default App
```

Для разработки системы безопасности мы будем использовать *контекст*, в котором хранятся данные компонента, доступные для его потомков. Например, маршрутизатор `BrowserRouter` с учетом контекста передает информацию о маршрутах находящимся в нем компонентам `Route`.

Мы создадим специальный контекст, называющийся `SecurityContext` (листинг 2.29).

Листинг 2.29. Код контекста SecurityContext

```
import React from 'react'
const SecurityContext = React.createContext({})
export default SecurityContext
```

По умолчанию наш контекст имеет значение пустого объекта. Нам нужен какой-либо способ, чтобы добавить в контекст функции для входа в систему и выхода из нее. Эта задача решается созданием компонента `SecurityProvider` (листинг 2.30).

Листинг 2.30. Код компонента SecurityProvider

```
import { useState } from 'react'
import SecurityContext from './SecurityContext'
const SecurityProvider = (props) => {
   const [loggedIn, setLoggedIn] = useState(false)
   return (
      <SecurityContext.Provider
         value={{
            login: (username, password) => {
  // Примечание для группы разработки:
  // Возможно, здесь нужен более высокий уровень безопасности...
               if (username === 'fred' && password === 'password') {
                  setLoggedIn(true)
               }
            },
            logout: () => setLoggedIn(false),
            loggedIn,
         }}
      >
         {props.children}
      </SecurityContext.Provider>
   )
}
export default SecurityProvider
```

В реальной системе код был бы значительно другим. Скорее всего, компонент для входа в систему и выхода из нее можно было бы создать, используя веб-службу или стороннюю систему безопасности. Но в нашем примере компонент `SecurityProvider` отслеживает, был ли выполнен вход, используя простое булево значение `loggedIn`. Этот компонент сохраняет в контексте следующие три элемента:

- функцию для входа в систему (`login`);
- функцию для выхода из системы (`logout`);
- булево значение, обозначающее, выполнен ли вход или нет (`loggedIN`).

Эти три элемента будут доступны любому компоненту, находящемуся в компоненте `SecurityProvider`. Чтобы обеспечить доступ к этим функциям любому компоненту внутри компонента `SecurityProvider`, добавим в код специальный хук `useSecurity` (листинг 2.31).

Листинг 2.31. Код хука `useSecurity`

```
import SecurityContext from './SecurityContext'
import { useContext } from 'react'
const useSecurity = () => useContext(SecurityContext)
export default useSecurity
```

Создав компонент `SecurityProvider`, нам нужно использовать его для защиты нескольких из существующих маршрутов приложения. Для этого создадим еще один компонент, `SecureRoute` (листинг 2.32).

Листинг 2.32. Код компонента `SecureRoute`

```
import Login from './Login'
import { Route } from 'react-router-dom'
import useSecurity from './useSecurity'
const SecureRoute = (props) => {
   const { loggedIn } = useSecurity()
   return (
      <Route {...props}>{loggedIn ? props.children : <Login />}</Route>
   )
}
export default SecureRoute
```

Компонент `SecureRoute` получает текущее состояние компонента `loggedIn` от компонента `SecurityContext`, используя для этого хук `useSecurity`. Если пользователь выполнил вход, то компонент `SecureRoute` отрисовывает содержимое маршрута. В противном случае отображается форма `LoginForm` для входа в систему[8].

Эта форма вызывает функцию `login`, которая при успешном входе в систему повторно отрисовывает компонент `SecureRoute`, а затем отображает защищенные данные.

Для совместного использования всех этих новых компонентов файл App.js модифицируется, как показано в листинге 2.33.

Листинг 2.33. Конечная версия содержимого файла App.js приложения

```
import './App.css'
import { BrowserRouter, Route, Switch } from 'react-router-dom'
```

[8] Здесь мы не приводим содержимое компонента `Login`, но весь его код доступен в репозитории GitHub этой книги.

```
import Public from './Public'
import Private1 from './Private1'
import Private2 from './Private2'
import Home from './Home'
import SecurityProvider from './SecurityProvider'
import SecureRoute from './SecureRoute'
function App() {
   return (
      <div className="App">
         <BrowserRouter>
            <SecurityProvider>
               <Switch>
                  <Route exact path="/">
                     <Home />
                  </Route>
                  <SecureRoute path="/private1">
                     <Private1 />
                  </SecureRoute>
                  <SecureRoute path="/private2">
                     <Private2 />
                  </SecureRoute>
                  <Route exact path="/public">
                     <Public />
                  </Route>
               </Switch>
            </SecurityProvider>
         </BrowserRouter>
      </div>
   )
}
export default App
```

Вся система маршрутизации заключается в оболочку из элемента `SecurityProvider`, делая функции `login()`, `logout()` и компонент `loggedIn` доступными для каждого компонента `SecureRoute`.

На рис. 2.12 показано отображение начальной страницы при исполнении этого приложения.

При выборе ссылки **Public Page** открывается соответствующая страница (рис. 2.13).

Рис. 2.12. Домашняя страница со ссылками на другие страницы приложения

Рис. 2.13. Просмотр общедоступной страницы Public Page возможен без входа в систему

Но при выборе ссылки **Private Page 1** открывается страница с формой для входа в систему (рис. 2.14).

Содержимое этой страницы (рис. 2.15) можно просмотреть только после входа в систему, введя имя пользователя *fred* и пароль *password*.

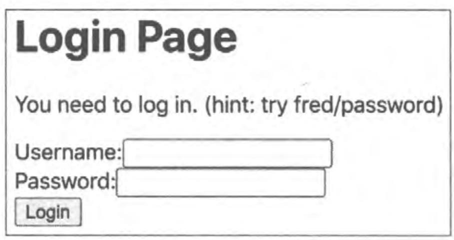


Рис. 2.14. Для просмотра содержимого страницы **Private Page 1** необходимо выполнить вход в систему

Рис. 2.15. Содержимое страницы с ограниченным доступом **Private Page 1** становится доступным после ввода правильного имени пользователя и пароля

Обсуждение

Настоящая безопасность может быть обеспечена только защищенными службами бэкенд-сервера. Но безопасные маршруты предотвращают возможность случайного попадания пользователей на страницы, которые не могут получать данные с сервера.

Для лучшей реализации компонента `SecurityProvider` нужно было бы воспользоваться каким-либо средством авторизации по стандарту OAuth или другими службами безопасности стороннего разработчика. Но отделение компонента `SecurityProvider` от пользовательского интерфейса службы безопасности (компоненты `Login` и `Logout`) и от основного приложения позволяет в дальнейшем изменять механизмы безопасности без необходимости модификации больших объемов кода самого приложения.

Чтобы увидеть поведение компонентов в процессе выполнения пользователями входа в систему и выхода из нее, можно всегда создать имитированную версию компонента `SecurityProvider` для использования в модульных тестах.

Исходный код данного рецепта можно загрузить на веб-сайте GitHub по адресу **https://oreil.ly/Kut73**.

ГЛАВА 3

Управление состоянием

При управлении состоянием в React требуется сохранение не только данных, но и зависимостей по данным. Зависимости являются неотъемлемым элементом для работы React, обеспечивая эффективное обновление страниц только тогда, когда это необходимо.

Вследствие этого управление зависимостями по данным — ключевой аспект управления состоянием в React. В этой главе мы рассмотрим большинство инструментов и методов, используемых для обеспечения эффективного управления зависимостями по данным.

Ключевое звено в первом рецепте — *преобразователь данных* (data reducer) — функция, которая принимает один объект или массив и возвращает его модифицированную версию. Эта простая концепция лежит в основе большей части управления состоянием в React. Мы рассмотрим нативное использование функций преобразования в React, а также применение библиотеки Redux для управления данными посредством преобразователей на уровне приложений.

Также мы рассмотрим *селекторные функции*, которые принимают предоставленные преобразователями состояния и возвращают их данные. Селекторные функции помогают нам игнорировать нерелевантные данные, тем самым значительно повышая производительность кода.

Попутно мы обратим внимание на простые способы проверки наличия подключения к Интернету, управления данными с форм и другие приемы и хитрости для обеспечения функционирования наших приложений должным образом.

3.1. Управление сложным состоянием посредством преобразователей

ЗАДАЧА

Многие компоненты React очень просты. Они не делают ничего больше, кроме как отрисовывают раздел HTML и, возможно, отображают несколько свойств.

Но некоторые компоненты могут быть довольно сложными, потенциально требующими управления несколькими частями внутреннего состояния. Рассмотрим, например, простую числовую игру-головоломку (рис. 3.1).

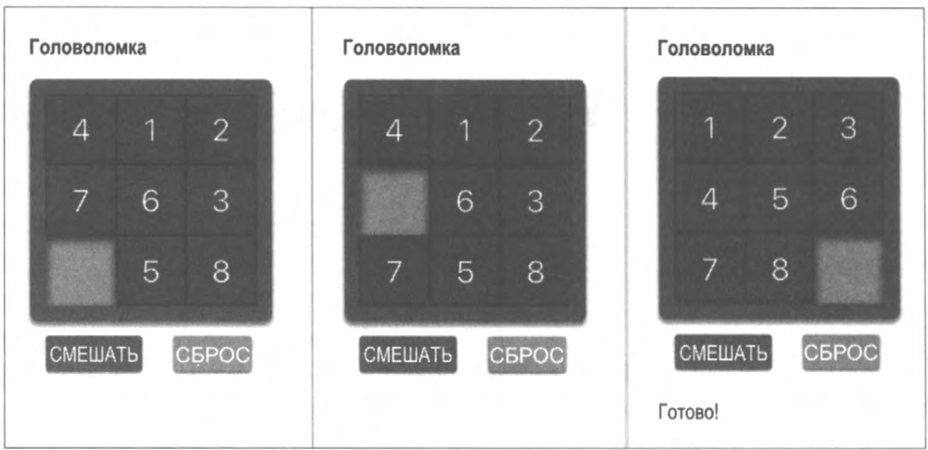


Рис. 3.1. Простая числовая игра-головоломка

Данный компонент отображает матрицу из нескольких пронумерованных плиток с одним свободным местом. Щелчок по плитке рядом со свободным местом позволяет переместить ее в это место. Суть игры состоит в расположении всех плиток в возрастающем порядке их номеров от 1 до 8.

Компонент отрисовывает лишь небольшой объем HTML-кода, но требует довольно сложной логики и сохранения данных. В частности, ему нужно будет запоминать местонахождение плиток, знать, можно или нет перемещать данную плитку и как переместить ее, знать, когда игра завершена, а также уметь выполнять другие действия, например перемешивать плитки для сброса игры.

Весь этот код вполне можно поместить в один компонент, но такой подход усложнит его тестирование. Для тестирования можно было бы воспользоваться библиотекой React Testing, но с учетом того, что программа будет выполнять очень небольшой объем отрисовки кода HTML, это, скорее всего, будет нерационально.

РЕШЕНИЕ

Для работы с компонентами со сложными внутренними состояниями или требующими сложных манипуляций состояний можно использовать *преобразователь* (reducer).

Преобразователь представляет собой функцию, которая принимает два параметра:

- объект или массив, представляющий данное состояние;
- действие, описывающее требующуюся модификацию этого состояния.

Функция возвращает модифицированную требуемым образом версию переданного ей состояния.

Параметр действия может быть любым, но обычно это объект со строковым атрибутом `type` и полезная нагрузка `payload` с дополнительной информацией. Строку `type` можно рассматривать как название команды, а нагрузку `payload` — как ее параметры.

Например, если пронумеровать позиции плиток от 0 (верхняя левая) до 8 (нижняя правая), то дать указание преобразователю переместить плитку из верхнего левого угла можно следующей командой:

```
{type: 'move', payload: 0}
```

Чтобы полностью определить внутреннее состояние нашей игры, нам требуется объект или массив, например простой массив строк:

```
['1', '2', '3', null, '5', '6', '7', '8', '4']
```

Это представляло бы следующее размещение плиток:

```
1 2 3
  5 6
7 8 4
```

Но более гибким подходом будет наличие объекта для представления состояния, в котором атрибут `items` представляет текущую организацию плиток:

```
{
  items: ['1', '2', '3', null, '5', '6', '7', '8', '4']
}
```

Этот способ предпочтительнее, поскольку он позволит нашему преобразователю возвращать значения других состояний, например завершена ли игра или нет:

```
{
  items: ['1', '2', '3', '4', '5', '6', '7', '8', null],
  complete: true
}
```

Таким образом, мы решили, какое действие выполнять (перемещение), и знаем, какой будет структура состояния. Это означает, что мы выполнили достаточный объем проектирования, чтобы создать тестовый код (листинг 3.1).

Листинг 3.1. Первый тестовый код

```
import reducer from './reducer'
describe('reducer', () => {
  it('should be able to move 1 down if gap below', () => {
    let state = {
      items: ['1', '2', '3', null, '5', '6', '7', '8', '4'],
    }
    state = reducer(state, { type: 'move', payload: 0 })
    expect(state.items).toEqual([
      null,
      '2',
      '3',
      '1',
      '5',
      '6',
      '7',
```

```
        '8',
        '4',
      ])
   })
   it('should say when it is complete', () => {
      let state = {
         items: ['1', '2', '3', '4', '5', '6', '7', null, '8'],
      }
      state = reducer(state, { type: 'move', payload: 8 })
      expect(state.complete).toBe(true)
      state = reducer(state, { type: 'move', payload: 5 })
      expect(state.complete).toBe(false)
   })
})
```

В первом тестовом сценарии мы передаем преобразователю расположение плиток в одном состоянии, а затем проверяем, что он возвращает плитки в новом состоянии.

Во втором тестовом сценарии мы выполняем два перемещения плиток, а затем проверяем на наличие атрибута `complete`, сообщающего, что игра завершена.

Теперь можно приступать к разработке кода самого преобразователя (листинг 3.2).

Листинг 3.2. Код преобразователя

```
function trySwap(newItems, position, t) {
   if (newItems[t] === null) {
      const temp = newItems[position]
      newItems[position] = newItems[t]
      newItems[t] = temp
   }
}

function arraysEqual(a, b) {
   for (let i = 0; i < a.length; i++) {
      if (a[i] !== b[i]) {
         return false
      }
   }
   return true
}
const CORRECT = ['1', '2', '3', '4', '5', '6', '7', '8', null]
function reducer(state, action) {
   switch (action.type) {
      case 'move': {
         const position = action.payload
         const newItems = [...state.items]
         const col = position % 3
```

```
        if (position < 6) {
          trySwap(newItems, position, position + 3)
        }
        if (position > 2) {
          trySwap(newItems, position, position - 3)
        }
        if (col < 2) {
          trySwap(newItems, position, position + 1)
        }
        if (col > 0) {
          trySwap(newItems, position, position - 1)
        }
        return {
          ...state,
          items: newItems,
          complete: arraysEqual(newItems, CORRECT),
        }
      }
      default: {
      throw new Error('Unknown action: ' + action.type)
      }
    }
}
export default reducer
```

На данном этапе наш преобразователь распознает единственное действие: `move` (перемещение). Полный код, который доступен в хранилище GitHub (**https://oreil.ly/q85H3**), также содержит действия для `shuffle` (перемешивание) и `reset` (сброс). В хранилище также доступен более обширный набор тестов (**https://oreil.ly/yRNyU**), которые мы использовали для создания кода в листинге 3.2.

Но весь этот код не содержит никаких компонентов React, а является чистым кодом JavaScript, поэтому его можно создать и протестировать в изоляции от внешнего мира.

Соблюдайте осторожность, и для представления нового состояния создавайте новый объект в преобразователе. Таким образом обеспечивается полная независимость каждого нового состояния от всех предыдущих.

Теперь настало время заключить наш преобразователь в оболочку компонента React, используя для этого хук `useReduce` (листинг 3.3).

Листинг 3.3. Заключение преобразователя в оболочку компонента React

```
import { useReducer } from 'react'
import reducer from './reducer'
import './Puzzle.css'
```

```
const Puzzle = () => {
  const [state, dispatch] = useReducer(reducer, {
    items: ['4', '1', '2', '7', '6', '3', null, '5', '8'],
  })
  return (
    <div className="Puzzle">
      <div className="Puzzle-squares">
        {state.items.map((s, i) => (
          <div
            className={`Puzzle-square ${
              s ? '' : 'Puzzle-square-empty'
            }`}
            key={`square-${i}`}
            onClick={() => dispatch({ type: 'move', payload: i })}
          >
            {s}
          </div>
        ))}
      </div>
      <div className="Puzzle-controls">
        <button
          className="Puzzle-shuffle"
          onClick={() => dispatch({ type: 'shuffle' })}
        >
          Shuffle
        </button>
        <button
          className="Puzzle-reset"
          onClick={() => dispatch({ type: 'reset' })}
        >
          Reset
        </button>
      </div>
      {state.complete && (
        <div className="Puzzle-complete">Complete!</div>
      )}
    </div>
  )
}
export default Puzzle
```

Хотя наш компонент головоломки выполняет довольно сложные операции, сам код React имеет сравнительно небольшой объем.

Хук `useReducer` принимает в качестве параметров функцию преобразователя и начальное состояние и возвращает двухэлементный массив:

- первый элемент массива содержит текущее состояние, возвращенное преобразователем;

- второй элемент массива содержит функцию `dispatch()`, которая позволяет отправлять действия преобразователю.

Плитки отображаются путем циклической обработки строки в массиве, предоставляемом в `state.items`.

При выборе пользователем плитки в положении `i` преобразователю отправляется следующая команда:

```
onClick={() => dispatch({type: 'move', payload: i})}
```

Компонент React не имеет никакого представления о том, как перемещать плитки. Он даже не знает, может ли он вообще перемещать их. Поэтому компонент отправляет действие преобразователю.

Если действие `move` перемещает плитку, компонент автоматически выполняет повторную отрисовку с плитками в новом положении. О завершении игры компонент узнает по значению `state.complete`:

```
state.complete && <div className='Puzzle-complete'>Complete!</div>
```

Мы также добавили код для двух кнопок для действий `shuffle` (перемешивание) и `reset` (сброс), который был опущен ранее, но содержится в полном коде в хранилище GitHub (**https://oreil.ly/WmZ18**).

Создав наш компонент, можно проверить его работу. При первоначальной загрузке компонента, отображается его исходное состояние (рис. 3.2).

Щелчок по плитке 7 перемещает ее на свободное место (рис. 3.3).

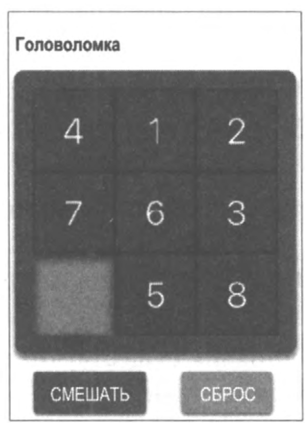


Рис. 3.2. Исходное состояние игры

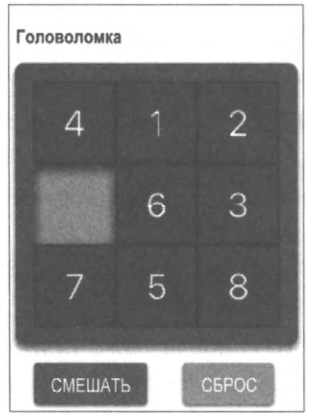


Рис. 3.3. Плитка 7 после перемещения на свободное место

При нажатии кнопки **Смешать** преобразователь располагает плитки в произвольном порядке. Результат такого действия показан на рис. 3.4.

А нажатие кнопки **Сброс** упорядочивает плитки по возрастанию номеров и отображает текст "Готово!".

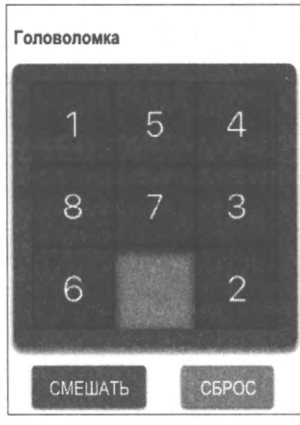


Рис. 3.4. Нажатие кнопки Shuffle перемешивает плитки

Рис. 3.5. При нажатии кнопки **Сброс** плитки упорядочиваются по возрастанию номеров

Все сложные подробности выполнения действий спрятаны в функции преобразователя, где их можно легко протестировать, а сам компонент прост и легок в обслуживании.

Обсуждение

Преобразователи — это средство для управления сложностью компонентов. Они обычно целесообразны в следующих случаях:

- при необходимости управлять большим объемом внутреннего состояния;
- при сложной логике управления внутренним состоянием компонента.

При наличии любого из этих обстоятельств преобразователь может значительно облегчить управление кодом.

Но подходите с осторожностью к использованию преобразователей для компонентов очень малого объема. Компоненты с простым состоянием и небольшим объемом логики, скорее всего, не нуждаются в дополнительной сложности преобразователя.

Кроме того, иногда даже при наличии сложного состояния существуют альтернативные подходы. Например, для захвата данных с формы и проверки их правильности может быть лучше создать компонент формы для проверки правильности данных (см. *раздел 3.3*).

Необходимо обеспечить отсутствие в преобразователе любых побочных эффектов. Например, избегайте сетевых вызовов для обновления сервера. При наличии в преобразователе побочных эффектов имеется высокая вероятность сбоя в работе. В режиме разработки React может иногда (подспудно) вызывать ваш преобразователь, чтобы убедиться в отсутствии побочных эффектов. Если вы обнаружите, что при отрисовке компонента React вызывает ваш код, содержащий преобразователь,

дважды, знайте, что это просто React проверяет на отсутствие нежелательного поведения кода.

С учетом всех этих оговорок преобразователи — это превосходный инструмент для борьбы со сложностью кода. Они являются неотъемлемой частью таких библиотек, как Redux, легко поддаются повторному использованию и объединению, упрощают компоненты, а также значительно облегчают тестирование кода React.

Исходный код данного рецепта можно загрузить на веб-сайте GitHub по адресу **https://oreil.ly/q85H3**.

3.2. Создание возможности "Отмена"

ЗАДАЧА

Часть потенциала насыщенных фреймворков JavaScript, таких как React, состоит в том, что создаваемые с их помощью веб-приложения могут быть очень похожими на приложения для настольных систем. Одна из распространенных возможностей настольных приложений — способность отменять выполненное действие. Некоторые нативные компоненты в приложениях React автоматически поддерживают функцию отмены. Например, если после редактирования данных в текстовом поле нажать клавиши <Cmd> или <Ctrl> + <Z>, то выполненные в тексте изменения будут отменены. Но нельзя ли распространить возможность отмены на пользовательские компоненты? Каким образом можно отслеживать изменения состояния, не создавая для этого больших объемов кода?

РЕШЕНИЕ

Если состоянием вашего компонента управляет функция преобразователя, то довольно общую возможность отмены можно реализовать при помощи функции преобразователя-отменщика.

Рассмотрим следующий фрагмент кода приложения головоломки из *раздела 3.1*:

```
const [state, dispatch] = useReducer(reducer, {
  items: ['4', '1', '2', '7', '6', '3', null, '5', '8'],
})
```

Этот код управляет размещением плиток в числовой игре-головоломке (рис. 3.6), используя функцию преобразования (называющуюся `reducer`) и исходное состояние.

При нажатии кнопки **Смешать** компонент обновляет состояние плиток, отправляя преобразователю действие `shuffle`:

```
<button className='Puzzle-shuffle'
  onClick={() => dispatch({type: 'shuffle'})}>Shuffle</button>
```

Дополнительная информация о том, что собой представляют преобразователи и когда следует их использовать, приводится в *разделе 3.1*.

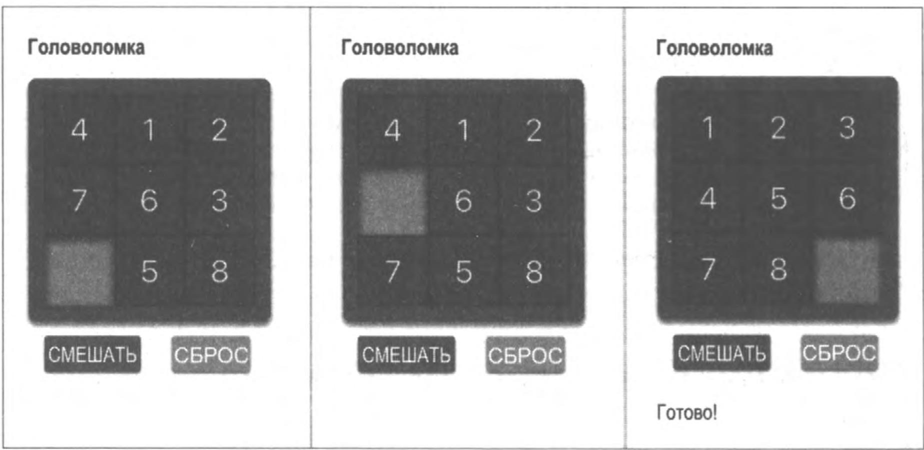


Рис. 3.6. Простая числовая игра-головоломка

Мы создадим новый хук, называющийся `useUndoReducer`, который будет прямой заменой хуку `useReducer`:

```
const [state, dispatch] = useUndoReducer(reducer, {
  items: ['4', '1', '2', '7', '6', '3', null, '5', '8'],
})
```

Этот хук снабдит наш компонент возможностью возвращаться назад во времени при нажатии кнопки **Отменить** (листинг 3.4).

Листинг 3.4. Код кнопки отмены

```
<button
  className="Puzzle-undo"
  onClick={() => dispatch({ type: 'undo' })}
>
  Undo
</button>
```

Добавление этого кода в компонент создает кнопку **Отменить** (рис. 3.7), нажатие которой отменяет последнее выполненное пользователем действие.

Но каким образом выполняется эта отмена? Хотя хук `useUndoReducer` сравнительно легко использовать, понять, как он работает, несколько труднее. Но в этом следует разобраться, чтобы можно было приспособить данный рецепт под свои требования.

Воспользуемся тем обстоятельством, что все преобразователи работают одинаково:

- действие определяет, что нужно сделать;
- после каждого действия преобразователь возвращает обновленное состояние;
- при вызове преобразователя не допускается никаких побочных эффектов.

Кроме того, преобразователи — это просто функции JavaScript, которые принимают в качестве параметров объект состояния и объект действия.

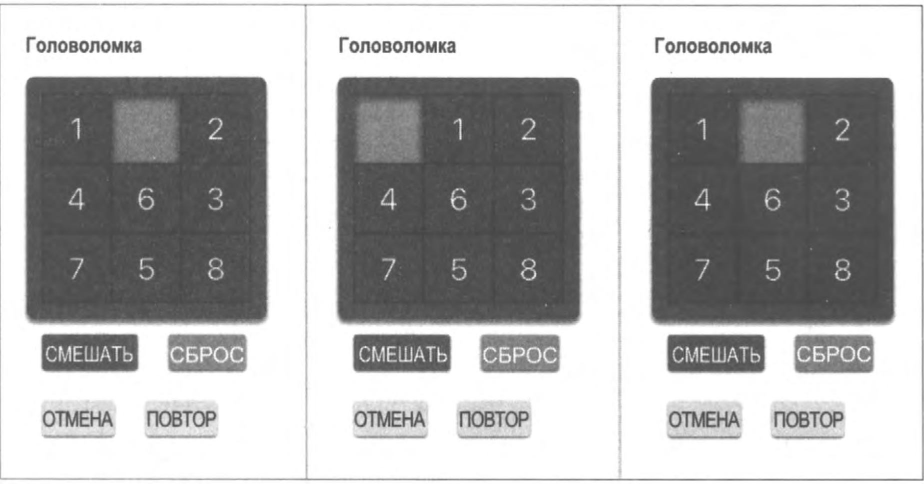


Рис. 3.7. Слева направо:
исходное состояние, перемещение плитки, отмена перемещения плитки

Поскольку способ работы преобразователя четко задан, можно создать новый преобразователь (преобразователь-отменщик), который облекает другую функцию преобразования. Этот преобразователь-отменщик будет служить посредником, передавая большинство действий внутреннему преобразователю, ведя историю всех предыдущих состояний. Если пользователь хочет отменить последнее действие, преобразователь-отменщик находит предшествующее этому действию состояние в своей истории и возвращает его, не вызывая для этого внутренний преобразователь.

Начнем с создания функции преобразователя высшего порядка, которая принимает в качестве параметра один преобразователь и возвращают другой (листинг 3.5).

Листинг 3.5. Код функции преобразователя высшего порядка

```
import lodash from 'lodash'
const undo = (reducer) => (state, action) => {
  let {
    undoHistory = [],
    undoActions = [],
    ...innerState
  } = lodash.cloneDeep(state)
  switch (action.type) {
    case 'undo': {
      if (undoActions.length > 0) {
        undoActions.pop()
        innerState = undoHistory.pop()
      }
      break
    }
```

```
      case 'redo': {
         if (undoActions.length > 0) {
            undoHistory = [...undoHistory, { ...innerState }]
            undoActions = [
               ...undoActions,
               undoActions[undoActions.length - 1],
            ]
            innerState = reducer(
               innerState,
               undoActions[undoActions.length - 1]
            )
         }
         break
      }
      default: {
         undoHistory = [...undoHistory, { ...innerState }]
         undoActions = [...undoActions, action]
         innerState = reducer(innerState, action)
      }
   }
   return { ...innerState, undoHistory, undoActions }
}
export default undo
```

Это довольно сложная функция, и будет полезным уделить некоторое время, чтобы разобраться, что она делает.

Она создает функцию преобразователя, которая отслеживает передаваемые ей действия и состояния. Предположим, что компонент игры посылает действия смешивания плиток. Наш преобразователь сначала проверит, не имеет ли действие тип `undo` (отменить) или `redo` (повторить). Если нет, то он передает действие `shuffle` внутреннему преобразователю, который управляет плитками в игре (рис. 3.8).

При передаче внутреннему преобразователю действия `shuffle` код отмены отслеживает существующее состояние и действие `shuffle`, добавляя их к `undoHistory` (исто-

Рис. 3.8. Преобразователь-отменщик передает большинство действий внутреннему преобразователю

рия отмен) и `undoActions` (отмена действий). Затем он возвращает состояние внутреннего преобразователя, `undoHistory` и `undoActions`.

Если компонент игры передает действие `undo` (отмена), то преобразователь-отменщик возвращает из истории для отмен `undoHistory` предыдущее состояние, полностью обходя собственный преобразователь игры (рис. 3.9).

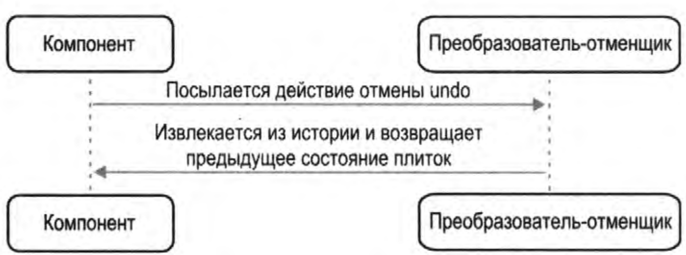


Рис. 3.9. Для действий отмены преобразователь-отменщик возвращает последнее состояние из истории

Теперь рассмотрим код самого хука `useUndoReducer` (листинг 3.6).

Листинг 3.6. Код хука `useUndoReducer`

```
import { useReducer } from 'react'
import undo from './undo'
const useUndoReducer = (reducer, initialState) =>
  useReducer(undo(reducer), initialState)
export default useUndoReducer
```

Код этого хука очень мал. В нем просто вызывается встроенный хук `useReducer`, но вместо прямой передачи этого преобразователя передается `undo(reducer)`. В результате наш компонент использует улучшенную версию предоставляемого нами преобразователя, которая может отменять и повторять действия.

В листинге 3.7 приведена обновленная версия компонента `Puzzle` с учетом этих модификаций. (Исходная версия приведена в *разделе 3.1*.)

Листинг 3.7. Модифицированная версия кода компонента `Puzzle`

```
import reducer from './reducer'
import useUndoReducer from './useUndoReducer'
import './Puzzle.css'
const Puzzle = () => {
  const [state, dispatch] = useUndoReducer(reducer, {
    items: ['4', '1', '2', '7', '6', '3', null, '5', '8'],
  })
  return (
    <div className="Puzzle">
      <div className="Puzzle-squares">
```

```
          {state.items.map((s, i) => (
            <div
              className={`Puzzle-square ${
                s ? '' : 'Puzzle-square-empty'
              }`}
              key={`square-${i}`}
              onClick={() => dispatch({ type: 'move', payload: i })}
            >
              {s}
            </div>
          ))}
        </div>
        <div className="Puzzle-controls">
          <button
            className="Puzzle-shuffle"
            onClick={() => dispatch({ type: 'shuffle' })}
          >
            Shuffle
          </button>
          <button
            className="Puzzle-reset"
            onClick={() => dispatch({ type: 'reset' })}
          >
            Reset
          </button>
        </div>
        <div className="Puzzle-controls">
          <button
            className="Puzzle-undo"
            onClick={() => dispatch({ type: 'undo' })}
          >
            Undo
          </button>
          <button
            className="Puzzle-redo"
            onClick={() => dispatch({ type: 'redo' })}
          >
            Redo
          </button>
        </div>
        {state.complete && (
          <div className="Puzzle-complete">Complete!</div>
        )}
      </div>
  )
}
export default Puzzle
```

Всё, чем отличается модифицированная версия компонента игры от исходной, — это заменой хука `useReducer` на хук `useUndoReducer`, а также парой дополнительных кнопок для вызова действий отмены и повтора.

Если теперь запустить приложение на исполнение и передвинуть некоторые плитки, то эти перемещения можно отменить по одному за раз, как показано на рис. 3.10.

Рис. 3.10. Использование хука useUndoReducer позволяет посылать действия отмены и повтора ходов

Обсуждение

Рассмотренная здесь функция преобразователя-отменщика будет работать с преобразователями, которые принимают и возвращают состояния в виде объектов. Для преобразователей, использующих массивы для управления состояниями, функцию отмены `undo` нужно переделать должным образом.

Поскольку наш преобразователь-отменщик сохраняет историю всех предыдущих состояний, то, скорее всего, не следует применять его для работы с состояниями с большими объемами данных или когда ему может потребоваться выполнить огромное количество изменений. Если же все-таки подобная необходимость возникнет, то полезно ограничить максимальный объем истории.

Кроме того, имейте в виду, что история сохраняется в памяти, и если пользователь перезагрузит страницу, то вся история пропадет. Эту проблему, однако, можно решить, сохраняя глобальное состояние в локальном хранилище при любых его изменениях.

Исходный код данного рецепта можно загрузить на веб-сайте GitHub по адресу **https://oreil.ly/Oz27A**.

3.3. Создание форм и проверка действительности их данных

ЗАДАЧА

В большинстве приложений React используются формы в той или иной степени, и во многих приложениях для их создания применяется импровизированный подход. Если ваше приложение разрабатывается командой, то может обнаружиться, что одни члены команды используют отдельные переменные состояния для управления полями форм. Другие же разработчики могут сохранять состояние формы в однозначном объекте. Такие объекты легко передавать в форму и извлекать из нее, но обновление каждого их поля может быть сопряжено с трудностями. Анализ достоверности данных поля часто ведет к образованию так называемого спагетти-кода, когда проверка данных с одних форм выполняется при их представлении, а данные с других форм проверяются динамически, как пользовательские типы. В одних формах сообщения проверки могут выводиться при первоначальной загрузке формы, в других — только после изменения полей пользователем.

Результатом такого разнообразия в дизайне может быть отрицательное восприятие пользователем приложения или отсутствие последовательного подхода к разработке кода. Исходя из нашего опыта работы с командами разработчиков React, формы и проверка достоверности их данных являются распространенными камнями преткновения для разработчиков.

РЕШЕНИЕ

Придать разработке форм определенный уровень целостности можно, создав компонент `SimpleForm`, который будет служить оболочкой для одного или нескольких компонентов поля ввода `InputField`. В листинге 3.8 приведен пример использования компонентов `SimpleForm` и `InputField`.

Листинг 3.8. Использование компонентов `SimpleForm` и `InputField`

```
import { useEffect, useState } from 'react'
import './App.css'
import SimpleForm from './SimpleForm'
import InputField from './InputField'
const FormExample0 = ({ onSubmit, onChange, initialValue = {} }) => {
   const [formFields, setFormFields] = useState(initialValue)
   const [valid, setValid] = useState(true)
   const [errors, setErrors] = useState({})
   useEffect(() => {
      if (onChange) {
         onChange(formFields, valid, errors)
      }
   }, [onChange, formFields, valid, errors])
```

```
    return (
        <div className="TheForm">
            <h1>Single field</h1>
            <SimpleForm
                value={formFields}
                onChange={setFormFields}
                onValid={(v, errs) => {
                    setValid(v)
                    setErrors(errs)
                }}
            >
                <InputField
                    name="field1"
                    onValidate={(v) =>
                        !v || v.length < 3 ? 'Too short!' : null
                    }
                />
                <button
                    onClick={() => onSubmit && onSubmit(formFields)}
                    disabled={!valid}
                >
                    Submit!
                </button>
            </SimpleForm>
        </div>
    )
}
export default FormExample0
```

Для отслеживания состояния формы используется один объект `formFields`. При изменении любого поля формы происходит вызов метода `onChange` компонента `SimpleForm`. Поле `field1` проверяется методом `onValidate`, и при любом изменении состояния проверки вызывается метод `onValid` компонента `SimpleForm`. Проверка выполняется только в случае взаимодействия пользователя с полем, в результате которого устанавливается флаг изменения поля.

На рис. 3.11 показана форма, выводимая в результате исполнения приложения.

Отслеживать значения отдельных полей нет надобности. Значения отдельных полей вместе с атрибутами, получаемыми из названий полей, сохраняются в объекте-значении формы. Всеми подробностями проверки занимается компонент `InputField`: он обновляет значение формы и решает, когда выводить сообщение об ошибке.

На рис. 3.12 показан пример несколько более сложной формы, в которой компонент `SimpleForm` применяется для проверки нескольких полей.

Прежде чем приступать к созданию компонентов `SimpleForm` и `InputField`, необходимо разобраться, как они будут взаимодействовать друг с другом. Компонент `InputField` должен будет сообщить компоненту `SimpleForm` об изменении своего зна-

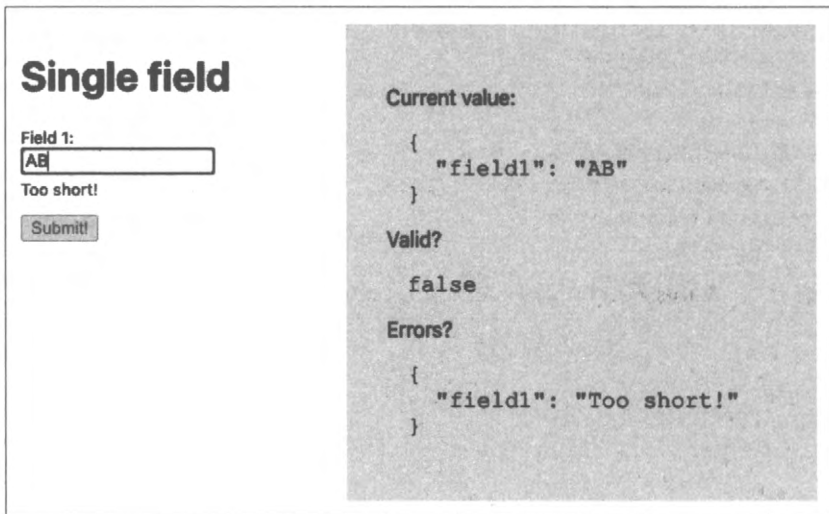


Рис. 3.11. Простая форма с проверкой действительности данных поля

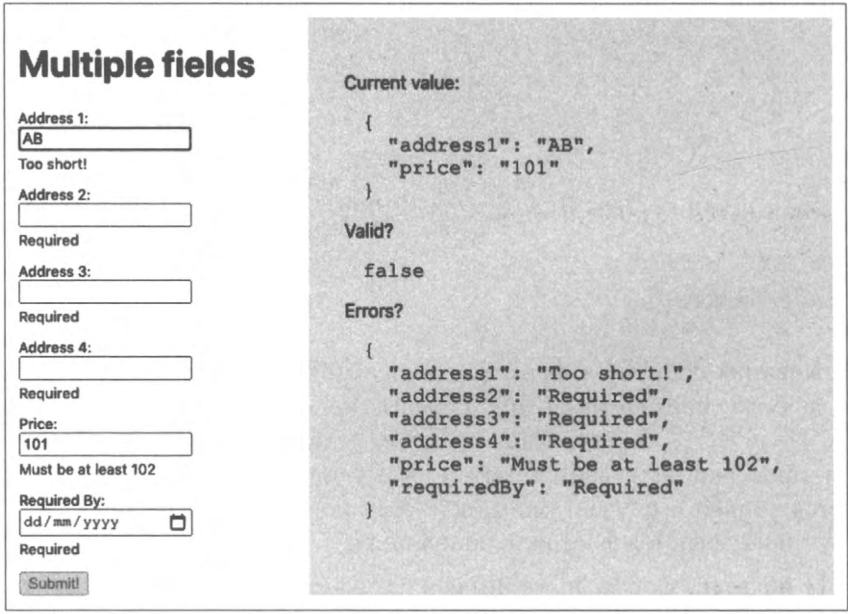


Рис. 3.12. Более сложная форма

чения, а также действительно ли это значение или нет. Все это компонент делает при помощи *контекста*.

Контекст представляет собой хранилище с областью видимости. Значения, сохраненные компонентом в контексте, видимы всем его подкомпонентам. Компонент `SimpleForm` создает контекст `FormContext` и хранит в нем набор функций обратного вызова, при помощи которых любой дочерний компонент может взаимодействовать с формой:

```
import { createContext } from 'react'
const FormContext = createContext({})
export default FormContext
```

Чтобы разобраться с работой компонента `SimpleForm`, рассмотрим его упрощенную версию, которая только отслеживает значения своих подкомпонентов, не заботясь на данном этапе об их проверке (листинг 3.9).

Листинг 3.9. Упрощенная версия компонента `SimpleForm`

```
import React, { useCallback, useEffect, useState } from 'react'
import './SimpleForm.css'
import FormContext from './FormContext'
function updateWith(oldValue, field, value) {
  const newValue = { ...oldValue }
  newValue[field] = value
  return newValue
}
const SimpleForm = ({ children, value, onChange, onValid }) => {
  const [values, setValues] = useState(value || {})
  useEffect(() => {
    setValues(value || {})
  }, [value])
  useEffect(() => {
    if (onChange) {
      onChange(values)
    }
  }, [onChange, values])
  let setValue = useCallback(
    (field, v) => setValues((vs) => updateWith(vs, field, v)),
    [setValues]
  )
  let getValue = useCallback((field) => values[field], [values])
  let form = {
    setValue: setValue,
    value: getValue,
  }
  return (
    <div className="SimpleForm-container">
      <FormContext.Provider value={form}>
        {children}
      </FormContext.Provider>
    </div>
  )
}
export default SimpleForm
```

В конечную версию компонента SimpleForm будет добавлен код для проверки отслеживания и вывода сообщений об ошибках, но пока проще разобраться с происходящим на примере такой урезанной версии компонента.

Отслеживание значений всех полей формы осуществляется в объекте состояния values. Форма создает две функции обратного вызова, getValue и setValue, и сохраняет их в контексте (в виде объекта form), где подкомпоненты смогут найти их. Чтобы сохранить форму form в контексте, дочерние компоненты помещаются в оболочку <FormContext.Provider>.

Обратите внимание на то, что мы обернули функции обратного вызова getValue и setValue в useCallback, что предотвращает создание компонентом новых версий этих функций при каждой отрисовке компонента SimpleForm.

При каждом вызове дочерним компонентом функции form.value() он получает текущее значение указанного поля. При вызове дочерним компонентом функции form.setValue() он обновляет это значение.

Теперь рассмотрим упрощенную версию компонента InputField, опять же без кода проверки, чтобы было легче разобраться с его работой (листинг 3.10).

Листинг 3.10. Упрощенная версия компонента InputField

```
import React, { useContext } from 'react'
import FormContext from './FormContext'
import './InputField.css'
const InputField = (props) => {
   const form = useContext(FormContext)
   if (!form.value) {
      return 'InputField should be wrapped in a form'
   }
   const { name, label, ...otherProps } = props
   const value = form.value(name)
   return (
      <div className="InputField">
         <label htmlFor={name}>{label || name}:</label>
            <input
               id={name}
               value={value || ''}
               onChange={(event) => {
                  form.setValue(name, event.target.value)
               }}
            {...otherProps}
         />{' '}
         {}
      </div>
   )
}
export default InputField
```

Компонент `InputField` извлекает объект `form` из контекста `FormContext`. Если он не может найти объект `form`, следовательно, этот объект не был помещен в оболочку компонента `SimpleForm`. Затем компонент отрисовывает поле `input`, устанавливая его значение равным значению, возвращенному функцией `form.value(имя_поля)`. При изменении значения поля компонент `InputField` отправляет новое значение функции `form.setValue(name, event.target.value)`.

Если требуется поле формы иного типа, нежели поле ввода, его можно заключить в оболочку какого-либо компонента наподобие рассмотренного здесь компонента `InputField`.

Код для проверки действительности вводимых значений во многом такой же, как и уже рассмотренный. Форма должна отслеживать, какие поля были изменены, а также значения каких полей недействительны, таким же образом, каким она отслеживает свое текущее значение в объекте состояния `values`. Затем форма должна передать обратные вызовы для `setDirty`, `isDirty` и `setInvalid`. Эти обратные вызовы используются дочерними полями при исполнении своего кода `onValidate`.

Окончательная версия кода компонента `SimpleForm`, включая код проверки действительности значений полей, приведена в листинге 3.11.

Листинг 3.11. Окончательная версия компонента `SimpleForm`

```
import { useCallback, useEffect, useState } from 'react'
import FormContext from './FormContext'
import './SimpleForm.css'
const SimpleForm = ({ children, value, onChange, onValid }) => {
   const [values, setValues] = useState(value || {})
   const [dirtyFields, setDirtyFields] = useState({})
   const [invalidFields, setInvalidFields] = useState({})
   useEffect(() => {
      setValues(value || {})
   }, [value])
   useEffect(() => {
      if (onChange) {
         onChange(values)
      }
   }, [onChange, values])
   useEffect(() => {
      if (onValid) {
         onValid(
            Object.keys(invalidFields).every((i) => !invalidFields[i]),
            invalidFields
         )
      }
   }, [onValid, invalidFields])
   const setValue = useCallback(
      (field, v) => setValues((vs) => ({ ...vs, [field]: v })),
```

```
    [setValues]
  )
  const getValue = useCallback((field) => values[field], [values])
  const setDirty = useCallback(
    (field) => setDirtyFields((df) => ({ ...df, [field]: true })),
    [setDirtyFields]
  )
  const getDirty = useCallback(
    (field) => Object.keys(dirtyFields).includes(field),
    [dirtyFields]
  )
  const setInvalid = useCallback(
    (field, error) => {
      setInvalidFields((i) => ({
        ...i,
        [field]: error ? error : undefined,
      }))
    },
    [setInvalidFields]
  )
  const form = {
    setValue: setValue,
    value: getValue,
    setDirty: setDirty,
    isDirty: getDirty,
    setInvalid: setInvalid,
  }
  return (
    <div className="SimpleForm-container">
      <FormContext.Provider value={form}>
        {children}
      </FormContext.Provider>
    </div>
  )
}
export default SimpleForm
```

А в листинге 3.12 приведена окончательная версия кода компонента `InputField`. Обратите внимание на то, что поле считается измененным (dirtied) при изменении его значения или при потере фокуса.

Листинг 3.12. Окончательная версия компонента InputField

```
import { useContext, useEffect, useState } from 'react'
import FormContext from './FormContext'
import './InputField.css'
```

```
const splitCamelCase = (s) =>
  s
    .replace(/([a-z0-9])([A-Z0-9])/g, '$1 $2')
    .replace(/^([a-z])/, (x) => x.toUpperCase())
const InputField = (props) => {
  const form = useContext(FormContext)
  const [error, setError] = useState('')
  const { onValidate, name, label, ...otherProps } = props
  let value = form.value && form.value(name)
  useEffect(() => {
    if (onValidate) {
      setError(onValidate(value))
    }
  }, [onValidate, value])
  const setInvalid = form.setInvalid
  useEffect(() => {
    if (setInvalid) {
      setInvalid(name, error)
    }
  }, [setInvalid, name, error])
  if (!form.value) {
    return 'InputField should be wrapped in a form'
  }
  return (
    <div className="InputField">
      <label htmlFor={name}>{label || splitCamelCase(name)}:</label>
      <input
        id={name}
        onBlur={() => form.setDirty(name)}
        value={value || ''}
        onChange={(event) => {
          form.setDirty(name)
          form.setValue(name, event.target.value)
        }}
        {...otherProps}
      />{' '}
      {
        <div className="InputField-error">
          {form.isDirty(name) && error ? error : <> </>}
        </div>
      }
    </div>
  )
}
export default InputField
```

Обсуждение

Но основе этого рецепта можно создавать много разных простых форм. Его также можно расширить для использования с любым компонентом React. Например, чтобы в компоненте `SimpleForm` реализовать календарь или поле выбора даты, весь компонент нужно всего лишь заключить в оболочку наподобие оболочки компонента `InputField`.

Но данный рецепт не дает возможности создавать формы в формах или в массивах форм. Чтобы поместить одну форму в другую, компонент `SimpleForm` можно модифицировать, чтобы он вел себя подобно компоненту `InputField`.

Исходный код данного рецепта можно загрузить на веб-сайте GitHub по адресу **https://oreil.ly/gU03F**.

3.4. Часы для измерения времени

ЗАДАЧА

Иногда приложение React должно выполнять определенное действие в назначенное время. Действием может быть простое отображение текущего времени, опрос сервера через регулярные интервалы времени или смена интерфейса при наступлении вечера. Но как заставить код выполнить отрисовку в результате изменения времени? Как избежать слишком частой отрисовки компонентов? И как реализовать все это без излишнего усложнения кода?

РЕШЕНИЕ

Для решения данной задачи создадим хук `useClock`, который предоставит нам доступ к отформатированной версии значения текущей даты и времени и автоматически обновит интерфейс при изменении времени. В листинге 3.13 приведен пример использования этого хука в приложении, а на рис. 3.13 показан результат исполнения этого приложения.

Листинг 3.13. Приложение с использованием хука useClock

```
import { useEffect, useState } from 'react'
import useClock from './useClock'
import ClockFace from './ClockFace'
import './Ticker.css'
const SimpleTicker = () => {
   const [isTick, setTick] = useState(false)
   const time = useClock('HH:mm:ss')
   useEffect(() => {
      setTick((t) => !t)
   }, [time])
```

```
  return (
    <div className="Ticker">
      <div className="Ticker-clock">
        <h1>Time {isTick ? 'Tick!' : 'Tock''}</h1>
        {time}
        <br />
        <ClockFace time={time} />
      </div>
    </div>
  )
}
export default SimpleTicker
```

Рис. 3.13. Отрисовка компонента `SimpleTicker` в течение периода длительностью в 3 секунды

Переменная `time` содержит значение текущего времени в формате `чч:мм:сс`. При изменении времени значение объекта состояния `isTick` меняется между булевыми значениями истина и ложь, и в зависимости от его конечного значения на экране отображается слово *Tick!* или *Tock*!. Также отображается текущее время в цифровом формате в виде обычного циферблата со стрелками (посредством компонента `ClockFace`).

Кроме формата даты и времени хук `useClock` также может принимать в качестве параметра число, задающее длительность интервала обновления в миллисекундах. В листинге 3.14 приведен соответствующий код, а на рис. 3.14 — результат его исполнения.

Листинг 3.14. Использование хука `useClock` для отображения длительности интервала

```
import { useEffect, useState } from 'react'
import useClock from './useClock'
import './Ticker.css'
```

```
const IntervalTicker = () => {
   const [isTick3, setTick3] = useState(false)
   const tickThreeSeconds = useClock(3000)
   useEffect(() => {
      setTick3((t) => !t)
   }, [tickThreeSeconds])
   return (
      <div className="Ticker">
         <div className="Ticker-clock">
            <h1>{isTick3 ? '3 Second Tick!' : '3 Second Tock!'}</h1>
            {tickThreeSeconds}
         </div>
      </div>
   )
}
export default IntervalTicker
```

Рис. 3.14. Объект `IntervalTicker` отрисовывает компонент каждые три секунды

Эта версия полезна в тех случаях, когда требуется выполнять какое-либо действие через регулярные интервалы времени, например опрашивать сетевую службу.

Рассмотрите возможность опроса сетевой службы с помощью проекта часов из *раздела 5.1*. Если хуку, осуществляющему сетевые вызовы, передать в качестве зависимости текущее значение часов, то опрос сети будет повторяться при каждом изменении этого значения.

Если хуку `useClock` передать числовой параметр, то он возвратит строку значения времени в формате ISO, например `2021-06-11T14:50:34.706`.

Для создания этого хука задействуем стороннюю библиотеку Moment.js (**https://momentjs.com**) для форматирования даты и времени:

```
$ npm install moment
```

Можно также выбрать какую-либо другую библиотеку, например Day.js (**https://day.js.org**). В таком случае можно легко переделать приведенный в листинге 3.15 код.

Листинг 3.15. Код для использования хука `useClock`

```
import { useEffect, useState } from 'react'
import moment from 'moment'
```

```
const useClock = (formatOrInterval) => {
  const format =
    typeof formatOrInterval === 'string'
      ? formatOrInterval
      : 'YYYY-MM-DDTHH:mm:ss.SSS'
  const interval =
    typeof formatOrInterval === 'number' ? formatOrInterval : 500
  const [response, setResponse] = useState(
    moment(new Date()).format(format)
  )
  useEffect(() => {
    const newTimer = setInterval(() => {
      setResponse(moment(new Date()).format(format))
    }, interval)
    return () => clearInterval(newTimer)
  }, [format, interval])
  return response
}
export default useClock
```

Из передаваемого хуку параметра `formatOrInterval` берутся формат даты и времени и требуемый интервал времени `interval`, после чего создается таймер посредством метода `setInterval()`. В результате значение переменной `response` будет устанавливаться на каждом интервале времени, задаваемом переменной `interval` (в миллисекундах). Когда переменной `response` задается новое значение времени, выполняется перерисовка любого компонента, взаимодействующего с хуком `useClock`.

Нам необходимо обязательно аннулировать все таймеры, которые больше не задействованы. Это можно сделать при помощи одной из возможностей хука `useEffect`. Если в конце кода хука `useEffect` возвратить функцию, то эта функция будет вызываться при следующем исполнении хука `useEffect`. Таким образом, мы можем с ее помощью аннулировать старый таймер, прежде чем создавать новый.

Если хуку `useClock` передать новый формат или интервал, то он аннулирует свой старый таймер и ответит, используя новый таймер.

Обсуждение

Этот рецепт — пример того, как посредством хуков можно решить несложную задачу простым способом. Код React (соответственно своему названию) реагирует на изменения в зависимостях. Хук `useClock` позволяет не беспокоиться о том, как исполнять код, например, каждую секунду, а просто разрабатывать код, который зависит от текущего времени и скрывает все трудные подробности создания таймеров, обновления состояния и обнуления таймеров.

Если хук `useClock` используется в компоненте несколько раз, тогда при изменении времени может быть выполнено несколько отрисовок. Например, если одни часы форматируют время в 12-часовом формате (04:45), а другие — в 24-часовом форма-

те (16:45), тогда при наступлении новой минуты будут выполнены две отрисовки вашего компонента. Но одна лишняя отрисовка раз в минуту вряд ли значительно ухудшит производительность.

Хук `useClock` можно также вставлять внутрь других хуков. Например, если создать хук `useMessages` для извлечения сообщений из сервера, то опрос сервера через регулярные интервалы времени можно выполнять, вызывая внутри этого хука хук `useClock`.

Исходный код данного рецепта можно загрузить на веб-сайте GitHub по адресу **https://oreil.ly/hohKK**.

3.5. Мониторинг состояния сетевого подключения

ЗАДАЧА

Предположим, что ваше приложение исполняется на сотовом телефоне, который может иногда терять подключение к сети, например при входе пользователя в метро. Каким образом можно проверить потерю подключения к сети? Существует ли несложный способ посредством React обновить интерфейс приложения, чтобы сообщить пользователю о проблеме или отключить возможности, требующие доступа к сети?

РЕШЕНИЕ

Эта задача решается созданием хука `useOnline`, который будет информировать нас о состоянии подключения к сети. Нам нужен код, который будет исполняться при потере и восстановлении браузером подключения к сети. К счастью, React содержит события `online` и `offline` уровня `window` и `body`, которые в точности реализуют требуемое нам действие. При срабатывании этих событий текущее состояние подключения к сети указывается соответствующим значением `true` или `false` свойства `navigator.onLine` (листинг 3.16)

Листинг 3.16. Использование событий `online` и `offline`

```
import { useEffect, useState } from 'react'
const useOnline = () => {
  const [online, setOnline] = useState(navigator.onLine)
  useEffect(() => {
    if (window.addEventListener) {
      window.addEventListener('online',()=>setOnline(true),false)
      window.addEventListener(
          'offline',
          () => setOnline(false),
          false
      )
```

```
    } else {
      document.body.ononline = () => setOnline(true)
      document.body.onoffline = () => setOnline(false)
    }
  }, [])
  return online
export default useOnline
```

Состояние подключения хука определяется значением переменной `online`. При первом исполнении хука (обратите внимание на пустой массив зависимостей) регистрируются прослушиватели (listeners) событий браузера подключения к сети и отключения от нее. Когда происходит какое-либо из этих событий, переменной `online` можно присвоить значение `true` или `false`. Если при этом изменяется текущее значение данной переменной, то выполняется отрисовка любого подключенного к данному хуку компонента.

В листинге 3.17 приведен код примера использования этого хука.

Листинг 3.17. Код с использованием хука `useOnline`

```
import useOnline from './useOnline'
import './App.css'
function App() {
  const online = useOnline()
  return (
    <div className="App">
      <h1>Network Checker</h1>
      <span>
        You are now....
        {online ? (
          <div className="App-indicator-online">ONLINE</div>
        ) : (
          <div className="App-indicator-offline">OFFLINE</div>
        )}
      </span>
    </div>
  )
}
export default App
```

При исполнении приложения на его странице отобразится наличие подключения к сети (online). Если отключиться, а затем снова подключиться к сети, то сообщения о состоянии подключения будут OFFLINE, а затем ONLINE, как показано на рис. 3.15.

Network Checker
You are now....ONLINE

Network Checker
You are now....OFFLINE

Network Checker
You are now....ONLINE

Рис. 3.15. Код исполняется при каждом отключении и подключении устройства к сети

Обсуждение

Важно отметить, что данный хук проверяет наличие подключения браузера к локальной сети, а не к сети Интернет или серверу. Чтобы проверить наличие подключения к серверу, потребуется дополнительный код.

Исходный код данного рецепта можно загрузить на веб-сайте GitHub по адресу **https://oreil.ly/9hkSA**.

3.6. Управление глобальным состоянием посредством библиотеки Redux

ЗАДАЧА

В других рецептах этой главы мы рассмотрели, как управлять сложным состоянием компонентов при помощи JavaScript функции, называющейся *преобразователем* (reducer). Преобразователи упрощают компоненты и облегчают тестирование логики программы.

Но что делать при необходимости повсеместного доступа к каким-либо данным, например к корзине покупателя?

РЕШЕНИЕ

Эту задачу можно решить, управляя глобальным состоянием приложения при помощи библиотеки Redux. В ней используются те же самые преобразователи, которые можно передавать функции `useReducer` платформы React, но только для управления одним объектом состояния для всего приложения. Кроме того, существует большое количество расширений Redux для решения распространенных задач программирования и для ускорения разработки и управления приложениями.

Первым делом установим библиотеку Redux:

```
$ npm install redux
```

Также нужно установить библиотеку React-Redux, которая значительно облегчает использование Redux с React:

```
$ npm install react-redux
```

Теперь все готово для создания приложения с корзиной покупателя (рис. 3.16).

При нажатии клиентом кнопки **Add to basket** (Положить в корзину) приложение добавляет данный продукт в корзину. При повторном нажатии этой кнопки количе-

ство данного товара в корзине обновляется. Корзина используется в нескольких местах приложения, следовательно, это хороший кандидат для реализации посредством Redux. В листинге 3.18 приведен код функции преобразователя для управления корзиной.

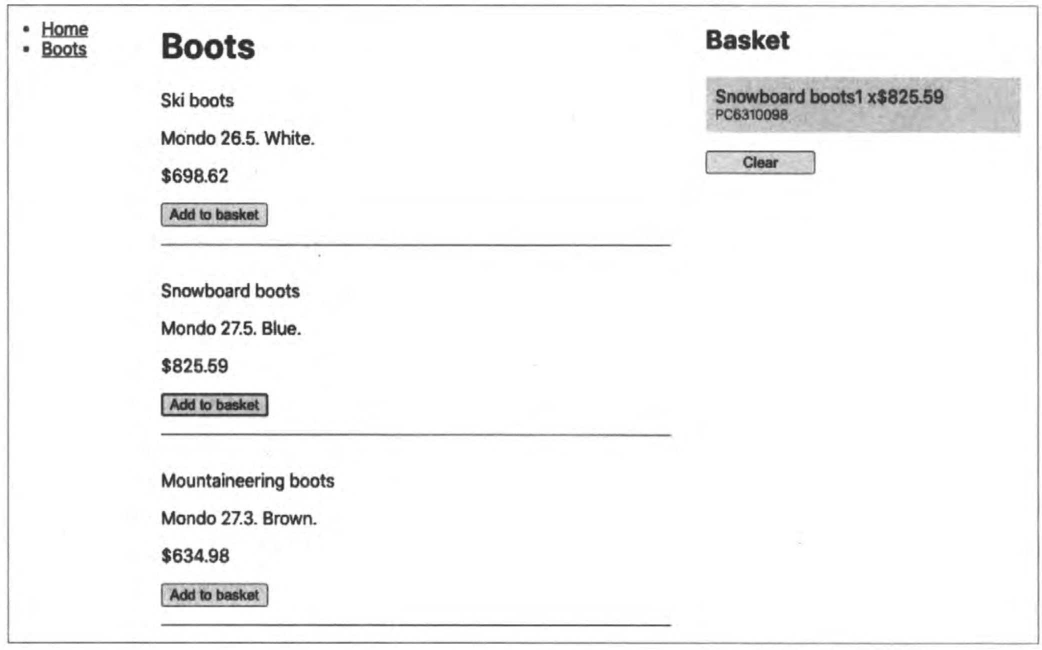


Рис. 3.16. Клиент добавляет товары в корзину через приложение

Листинг 3.18. Функция преобразователя для управления корзиной

```
const reducer = (state = {}, action = {}) => {
   switch (action.type) {
      case 'buy': {
         const basket = state.basket ? [...state.basket] : []
         const existing = basket.findIndex(
            (item) => item.productId === action.payload.productId
         )
         if (existing !== -1) {
            basket[existing].quantity = basket[existing].quantity + 1
         } else {
            basket.push({ quantity: 1, ...action.payload })
         }
         return {
            ...state,
            basket,
         }
      }
```

```
      case 'clearBasket': {
         return {
            ...state,
            basket: [],
         }
      }
      default:
      return { ...state }
   }
}
export default reducer
```

В данном случае мы создаем только один преобразователь. Для приложений большего размера целесообразно разбить преобразователь на несколько меньших преобразователей, которые можно объединить посредством функции Redux `combineReducers()` **(https://oreil.ly/lVh7x).**

Функция преобразователя реагирует на действия покупки `buy` и очистки корзины `clearBasket`. Действие `buy` или добавляет новый товар в корзину, или обновляет количество товара с совпадающим номером `productId`, уже находящегося в корзине. Действие `clearBasket` обнуляет массив корзины.

Имея в своем распоряжении функцию преобразователя, создадим с ее помощью *хранилище* (store) Redux, которое будет центральным репозиторием для общего состояния приложения. Для этого добавим следующий фрагмент кода в какой-либо компонент высшего уровня, например файл App.js:

```
import { createStore } from 'redux'
import reducer from './reducer'
const store = createStore(reducer)
```

В приложении хранилище должно быть доступным глобально, поэтому нужно добавить его в контекст компонентов, которым оно может потребоваться. Это можно сделать посредством компонента `Provider` библиотеки React Redux:

```
<Provider store={store}>
   Все перечисленные здесь компоненты имеют доступ к хранилищу
</Provider>
```

В листинге 3.19 приведен фрагмент файла reducer.js из примера приложения, полный код которого доступен в репозитории GitHub для этой книги (**https://oreil.ly/j90xI**).

Листинг 3.19. Фрагмент файла reducer.js

```
const reducer = (state = {}, action = {}) => {
   switch (action.type) {
      case 'buy': {
         const basket = state.basket ? [...state.basket] : []
```

```
        const existing = basket.findIndex(
          (item) => item.productId === action.payload.productId
        )
        if (existing !== -1) {
          basket[existing].quantity = basket[existing].quantity + 1
        } else {
          basket.push({ quantity: 1, ...action.payload })
        }
        return {
        ...state,
        basket,
      }
    }
    case 'clearBasket': {
      return {
        ...state,
        basket: [],
      }
    }
    default:
      return { ...state }
  }
}
export default reducer
```

Теперь хранилище доступно для наших компонентов, но как же им пользоваться? Библиотека React Redux обеспечивает доступ к хранилищу через хуки. Например, считать содержимое глобального состояния можно при помощи хука `useSelector`:

```
const basket = useSelector((state) => state.basket)
```

Хук `useSelector` принимает в качестве параметра функцию, из которой затем можно извлечь требуемую часть общего состояния. Преобразователи работают довольно эффективно и вызывают повторную отрисовку компонентов только при изменении интересующего нас состояния.

Отправить действие в центральное хранилище можно посредством хука `useDispatch`:

```
const dispatch = useDispatch()
```

Итогом будет возврат функции `dispatch`, при помощи которой можно отправлять действия в хранилище:

```
dispatch({ type: 'clearBasket' })
```

Принцип работы этих хуков основан на извлечении хранилища из текущего контекста. При попытке исполнить приложение, не содержащее компонент `Provider`, или исполнить хук `useSelector` или `useDispatch` вне его контекста, будет выдано сообщение об ошибке, как показано на рис. 3.17.

Error: could not find react-redux context value; please ensure the component is wrapped in a <Provider> ×

```
▶ 2 stack frames were collapsed.

./src/Basket.js/__webpack_exports__.default
src/Basket.js:7

   4 | import "./Basket.css";
   5 |
   6 | export default () => {
>  7 |     const basket = useSelector(state => state.basket);
   8 |     const dispatch = useDispatch();
   9 |
  10 |     return <div className='Basket'>

View compiled

▶ 16 stack frames were collapsed.

./src/index.js
src/index.js:7

   4 | import App from './App';
   5 | import * as serviceWorker from './serviceWorker';
```

Рис. 3.17. Сообщение об ошибке, возвращаемое при попытке исполнить приложение без компонента `Provider`

В листинге 3.20 приведен полный код компонента `Basket`, который считывает и очищает корзину покупателя на уровне всего приложения.

Листинг 3.20. Полный код компонента Basket

```
import { useDispatch, useSelector } from 'react-redux'
import './Basket.css'
const Basket = () => {
  const basket = useSelector((state) => state.basket)
  const dispatch = useDispatch()
  return (
    <div className="Basket">
      <h2>Basket</h2>
      {basket && basket.length ? (
        <>
          {basket.map((item) => (
            <div className="Basket-item">
              <div className="Basket-itemName">{item.name}</div>
              <div className="Basket-itemProductId">
                {item.productId}
              </div>
```

```
                <div className="Basket-itemPricing">
                    <div className="Basket-itemQuantity">
                        {item.quantity}
                    </div>
                <div className="Basket-itemPrice">{item.price}</div>
                </div>
            </div>
        ))}
        <button onClick={() => dispatch({type: 'clearBasket'})}>
            Clear
        </button>
    </>
) : (
    'Empty'
)}
</div>
)
}
export default Basket
```

Чтобы продемонстрировать добавление товаров в корзину, в листинге 3.21 приведен код компонента `Boots`, позволяющий клиенту купить товары определенного типа.

Листинг 3.21. Код компонента `Boots`

```
import { useDispatch } from 'react-redux'
import './Boots.css'
const products = [
    {
        productId: 'BE8290004',
        name: 'Ski boots',
        description: 'Mondo 26.5. White.',
        price: 698.62,
    },
    {
        productId: 'PC6310098',
        name: 'Snowboard boots',
        description: 'Mondo 27.5. Blue.',
        price: 825.59,
    },
    {
        productId: 'RR5430103',
        name: 'Mountaineering boots',
        description: 'Mondo 27.3. Brown.',
        price: 634.98,
    },
    ]
```

```
  const Boots = () => {
    const dispatch = useDispatch()
    return (
      <div className="Boots">
        <h1>Boots</h1>
        <dl className="Boots-products">
          {products.map((product) => (
            <>
              <dt>{product.name}</dt>
            <dd>
              <p>{product.description}</p>
              <p>${product.price}</p>
              <button
                onClick={() =>
                  dispatch({ type: 'buy', payload: product })
                }
              >
                Add to basket
              </button>
            </dd>
          </>
        ))}
      </dl>
    </div>
  )
}
export default Boots
```

Эти два компонента могут появляться в совершенно разных местах дерева компонентов, но используют одно и то же хранилище Redux. При добавлении продукта в корзину компонент `Basket` автоматически обновляет ее состояние, отражая это изменение (рис. 3.18).

Обсуждение

Разработчики часто применяют библиотеку Redux совместно с фреймворком React. Долгое время казалось, что почти каждое приложение React содержало Redux по умолчанию. Но, скорее всего, такое использование Redux было избыточным и часто неуместным. Нам приходилось видеть проекты, в которых было полностью запрещено локальное состояние и общее состояние определялось через Redux. Мы считаем такой подход ошибочным. Средства Redux предназначены для управления центральным состоянием приложения, а не состоянием отдельных компонентов. Если данные представляют интерес только для одного компонента или его подкомпонентов, то, скорее всего, их не следует сохранять в Redux.

Но если необходимо управление каким-либо глобальным состоянием приложения, тогда Redux будет предпочтительным инструментом.

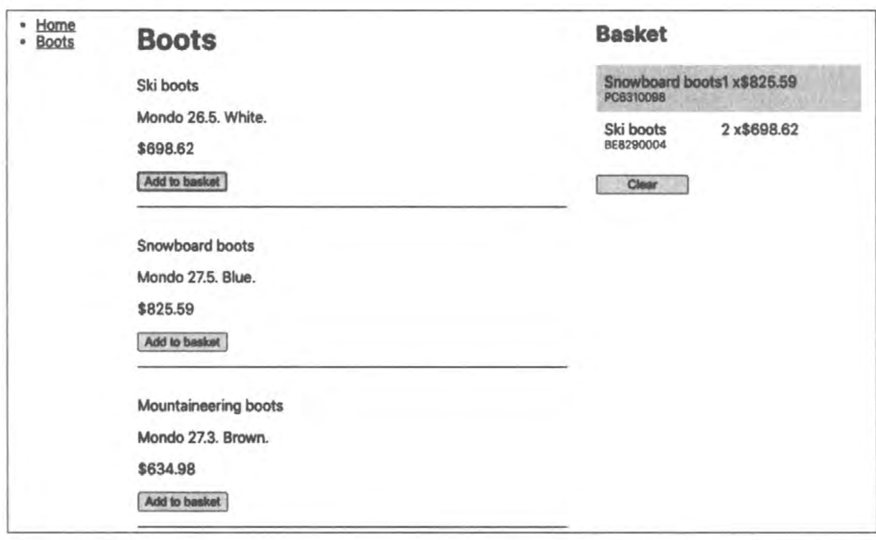


Рис. 3.18. Хуки Redux-React обеспечивают обновление отрисовки при добавлении товара в корзину

Исходный код данного рецепта можно загрузить на веб-сайте GitHub по адресу **https://oreil.ly/j90xI**.

3.7. Сохранение состояния при обновлении страниц посредством Redux Persist

ЗАДАЧА

Библиотека Redux предоставляет отличные средства для централизованного управления состоянием приложения. Но она имеет один небольшой недостаток: перезагрузка страницы вызывает обнуление всего состояния (рис. 3.19).

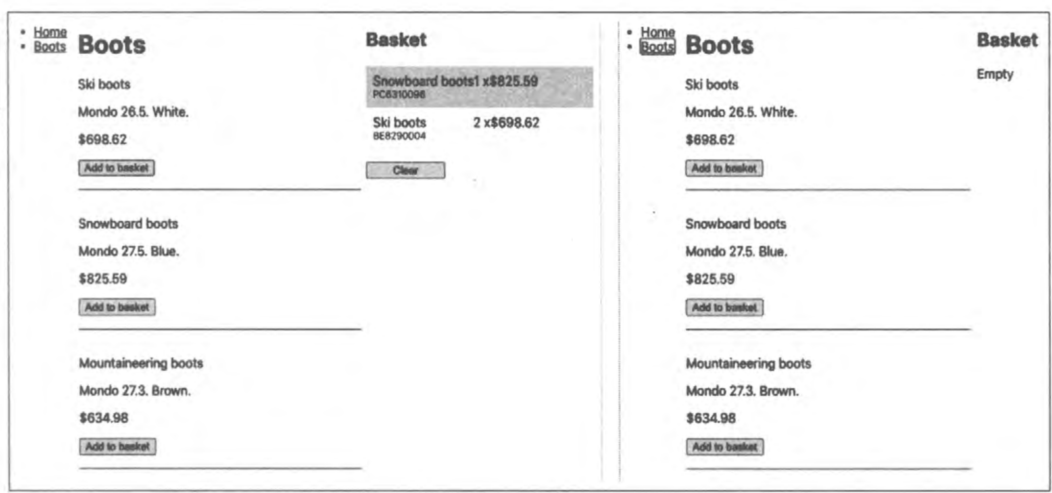


Рис. 3.19. Состояние Redux (слева) теряется при перезагрузке страницы (справа)

Такое поведение объясняется тем, что состояние Redux хранится в памяти. Можно ли каким-либо образом предотвратить потерю состояния?

РЕШЕНИЕ

Эта задача решается путем сохранения в локальном хранилище копии состояния Redux при помощи библиотеки Redux Persist. Первым делом установим эту библиотеку:

```
$ npm install redux-persist
```

Затем нужно создать постоянный преобразователь, заключив в него наш существующий преобразователь (листинг 3.22).

Листинг 3.22. Постоянный преобразователь

```
import storage from 'redux-persist/lib/storage'
const persistConfig = {
   key: 'root',
   storage,
}
const persistedReducer = persistReducer(persistConfig, reducer)
```

В переменной `storage` указывается, где будет храниться состояние Redux: по умолчанию в локальном хранилище `localStorage`. А константа `persistConfig` уточняет, где именно в локальном хранилище `localStorage` будет храниться наше состояние: в элементе `persist:root`. При изменении состояния Redux постоянный преобразователь `persistedReducer` будет сохранять копию состояния посредством метода `localStorage.setItem(' persist: root', ...)`. Теперь нам нужно создать хранилище Redux, содержащее преобразователь `persistedReducer`:

```
const store = createStore(persistedReducer)
```

Нам необходимо вставить код Redux Persist между хранилищем Redux и кодом, который обращается к нему. Эта задача решается посредством компонента `PersistGate` (листинг 3.23).

Листинг 3.23. Код компонента `PersistGate`

```
import { PersistGate } from 'redux-persist/integration/react'
import { persistStore } from 'redux-persist'
const persistor = persistStore(store)
...
<Provider store={store}>
   <PersistGate loading={<div>Loading...</div>} persistor={persistor}>
      Components live in here
   </PersistGate>
</Provider>
```

Компонент PersistGate должен находиться внутри метода Redux Provider и снаружи компонентов, которые будут использовать Redux. Этот компонент будет отслеживать состояние Redux и при его потере перезагружать его из локального хранилища localStorage. Загрузка данных состояния может занимать некоторое время, поэтому для индикации кратковременной занятости пользовательского интерфейса компоненту PersistGate можно передать компонент loading, например какой-либо анимированный элемент типа вращающегося значка занятости компьютера. Компонент loading будет отображаться вместо его дочерних элементов, пока выполняется загрузка состояния Redux. При ненадобности компонент loading можно отключить, присвоив ему значение null.

В листинге 3.24 приведена окончательная версия кода приложения из файла App.js.

Листинг 3.24. Окончательная версия кода приложения

```
import { BrowserRouter, Route, Switch } from 'react-router-dom'
import { Provider } from 'react-redux'
import { createStore } from 'redux'
import Menu from './Menu'
import Home from './Home'
import Boots from './Boots'
import Basket from './Basket'
import './App.css'
import reducer from './reducer'
import { persistStore, persistReducer } from 'redux-persist'
import { PersistGate } from 'redux-persist/integration/react'
import storage from 'redux-persist/lib/storage'
const persistConfig = {
   key: 'root',
   storage,
}
const persistedReducer = persistReducer(persistConfig, reducer)
const store = createStore(persistedReducer)
const persistor = persistStore(store)
function App() {
   return (
      <div className="App">
         <Provider store={store}>
            <PersistGate
               loading={<div>Loading...</div>}
               persistor={persistor}
            >
               <BrowserRouter>
                  <Menu />
                  <Switch>
                     <Route exact path="/">
                        <Home />
                     </Route>
```

```
                <Route path="/boots">
                    <Boots />
                </Route>
            </Switch>
            <Basket />
        </BrowserRouter>
      </PersistGate>
    </Provider>
  </div>
 )
}
export default App
```

Рис. 3.20. Состояние Redux до (вверху) и после (внизу) перезагрузки страницы

Теперь при перезагрузке страницы состояние Redux сохраняется, как показано на рис. 3.20.

Обсуждение

Библиотека Redux Persist предоставляет простой способ сохранения состояния Redux при перезагрузке страниц. При этом следует быть осторожным, чтобы не превысить максимальный предел хранилища `localStorage` большими объемами данных Redux. Этот предел зависит от конкретного браузера, но обычно составляет около 10 МБайт. В случаях, когда объем данных Redux превышает этот размер, следует рассмотреть возможность сохранения некоторой части данных на сервере.

Исходный код данного рецепта можно загрузить на веб-сайте GitHub по адресу **https://oreil.ly/K8U5J**.

3.8. Вычисление производного состояния посредством Reselect

ЗАДАЧА

Во многих случаях, прежде чем отображать данные состояния приложения, извлеченные во внешний объект при помощи Redux, их нужно подвергнуть той или иной обработке. Вернемся к приложению, которое мы рассматривали в нескольких предыдущих рецептах этой главы (рис. 3.21).

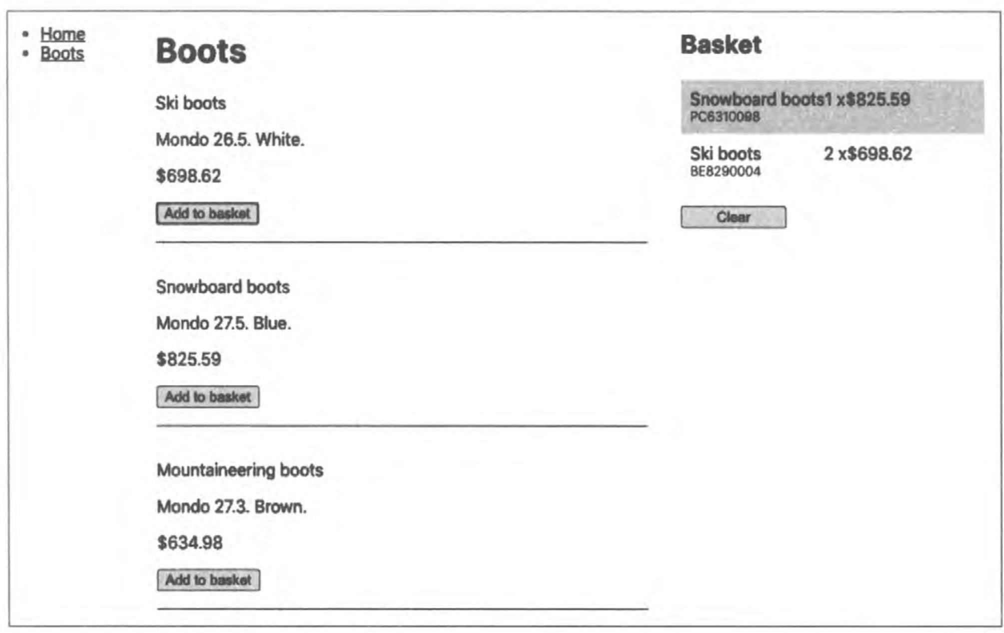


Рис. 3.21. Как наилучшим образом вычислить общую стоимость товаров в корзине и налог на них?

Предположим, что нам нужно вычислить общую стоимость товаров в корзине, а затем рассчитать налог с продаж на эту сумму. Данную задачу можно было бы решить при помощи функции JavaScript, обрабатывающей все товары в корзине и вычисляющей требуемую информацию. Но такая функция должна будет повторно выполнять все вычисления при каждой отрисовке корзины. Есть ли какой-либо способ вычислить производные значения по состоянию и обновлять вычисления только при изменении состояния?

РЕШЕНИЕ

Данную задачу можно решить при помощи библиотеки Reselect, специально созданной разработчиками Redux для эффективного вычисления значений по объектам состояний.

Библиотека Reselect создает селекторные функции. *Селекторная функция* (или просто *селектор*) принимает объект состояния в качестве параметра и возвращает обработанную версию этого объекта.

В *разделе 3.6* мы уже один раз встречались с селектором, при возврате текущего состояния корзины из центрального состояния Redux:

```
const basket = useSelector((state) => state.basket)
```

Здесь селекторная функция `state =>state.basket` извлекает некое значение из объекта состояния. Библиотека Reselect создает высокоэффективные селекторные функции, которые могут кешировать свои результаты при отсутствии изменений в состоянии, от которого они зависят.

Первым делом установим эту библиотеку:

```
$ npm install reselect
```

Теперь мы можем создать селекторную функцию, которая будет выполнять следующие операции:

- подсчитывать общее количество товаров в корзине;
- вычислять общую стоимость всех товаров.

Назовем эту функцию `summarizer`. Прежде чем рассматривать подробности написания этой функции, создадим тест, который покажет, что она должна делать (листинг 3.25).

Листинг 3.25. Тест для демонстрации требований к функции `summarizer`

```
it('should be able to handle multiple products', () => {
   const actual = summarizer({
      basket: [
         { productId: '1234', quantity: 2, price: 1.23 },
         { productId: '5678', quantity: 1, price: 1.5 },
      ],
   })
   expect(actual).toEqual({ itemCount: 3, cost: 3.96 })
})
```

Таким образом, при передаче этой функции объекта состояния она суммирует количество и стоимость товаров и возвращает объект, содержащий соответствующие переменные `itemCount` и `cost`.

Посредством библиотеки Reselect селекторную функцию `summarizer` можно создать, как показано в листинге 3.26.

Листинг 3.26. Селекторная функция `summarizer`

```
import { createSelector } from 'reselect'
const summarizer = createSelector(
   (state) => state.basket || [],
   (basket) => ({
      itemCount: basket.reduce((i, j) => i + j.quantity, 0),
      cost: basket.reduce((i, j) => i + j.quantity * j.price, 0),
   })
)
export default summarizer
```

Селекторная функция создается с помощью функции `createSelector` на основе других селекторных функций. Все параметры, передаваемые этой функции, за исключением последнего, должны быть селекторными функциями. В данном случае мы передаем только один параметр:

```
(state) => state.basket || []
```

Этот код извлекает компонент корзины из объекта состояния.

Последний параметр, передаваемый функции `createSelector`, называется *объединителем* (combiner). Это функция, которая вычисляет новое значение на основании результатов предшествующих селекторов (листинг 3.27).

Листинг 3.27. Код функции-объединителя

```
(basket) => ({
   itemCount: basket.reduce((i, j) => i + j.quantity, 0),
   cost: basket.reduce((i, j) => i + j.quantity * j.price, 0),
})
```

Значение `basket` получается в результате обработки состояния первым селектором.

Но чем объяснить такой необычный способ создания функций? Разве это ненамного сложнее, чем просто создать функцию JavaScript вручную, не передавая все эти функции другим функциям?

Ответить на эти вопросы можно одним словом — эффективность. Селекторы перевычисляют свои значения только тогда, когда это требуется. Объекты состояния могут иметь сложную структуру и десятки атрибутов. Но нас интересует только содержимое атрибута `basket`, и мы не хотим перевычислять стоимость при изменении значения каких-либо других атрибутов.

Функция summarizer решает, когда возвращаемое ею значение, скорее всего, было изменено. Предположим, что в первом вызове она вычисляет значения itemCount и value следующим образом:

```
{itemCount: 3, cost: 3.96}
```

Затем пользователь исполняет некоторое число команд, например обновляет настройки своих предпочтений, отправляет кому-то сообщение, добавляет несколько товаров в свой список покупок и т. п.

Каждое из этих действий может вызвать обновление глобального состояния приложения. Но при следующем вызове функции summarizer она возвратит из кеша ранее созданное значение:

```
{itemCount: 3, cost: 3.96}
```

Почему? Потому что она знает, что это значение зависит *только* от значения basket в глобальном состоянии. Если данное значение не изменилось, то нет надобности перевычислять возвращаемое значение.

Поскольку библиотека Reselect позволяет создавать селекторные функции из других селекторных функций, можно создать еще одну селекторную функцию, taxer, для вычисления налога с продаж (листинг 3.28).

Листинг 3.28. Функция taxer для вычисления налога с продаж

```
import { createSelector } from 'reselect'
import summarizer from './summarizer'
const taxer = createSelector(
   summarizer,
   (summary) => summary.cost * 0.07
)
export default taxer
```

Селектор taxer обрабатывает значение cost, возвращаемое функцией summarizer, умножая его на 0,07, т. е. вычисляет 7% от общей стоимости товаров в корзине. Если общая стоимость товаров в корзине не изменилась, то нет надобности обновлять результат функции taxer.

Функции summarizer и taxer можно использовать внутри компонентов, точно так же, как и любую другую селекторную функцию (листинг 3.29).

Листинг 3.29. Использование функций summarizer и taxer в компоненте

```
import { useDispatch, useSelector } from 'react-redux'
import './Basket.css'
import summarizer from './summarizer'
import taxer from './taxer'
const Basket = () => {
   const basket = useSelector((state) => state.basket)
```

```
    const { itemCount, cost } = useSelector(summarizer)
    const tax = useSelector(taxer)
    const dispatch = useDispatch()
    return (
      <div className="Basket">
        <h2>Basket</h2>
        {basket && basket.length ? (
          <>
            {basket.map((item) => (
              <div className="Basket-item">
               <div className="Basket-itemName">{item.name}</div>
               <div className="Basket-itemProductId">
                   {item.productId}
                 </div>
                 <div className="Basket-itemPricing">
                   <div className="Basket-itemQuantity">
                     {item.quantity}
                   </div>
                <div className="Basket-itemPrice">{item.price}</div>
                </div>
               </div>
            ))}
            <p>{itemCount} items</p>
            <p>Total: ${cost.toFixed(2)}</p>
            <p>Sales tax: ${tax.toFixed(2)}</p>
            <button onClick={() => dispatch({ type: 'clearBasket' })}>
              Clear
            </button>
          </>
        ) : (
          'Empty'
        )}
      </div>
    )
}
export default Basket
```

При исполнении кода теперь внизу корзины покупателя отображается общая стоимость находящихся в ней товаров, которая будет обновляться при добавлении в корзину новых покупок (рис. 3.22).

Обсуждение

На первый взгляд селекторные функции могут казаться сложными и трудными для понимания. Но, уделив некоторое время, чтобы разобраться с ними, можно понять, что ничего сложного в них нет. В них нет ничего, присущего только Redux, и их

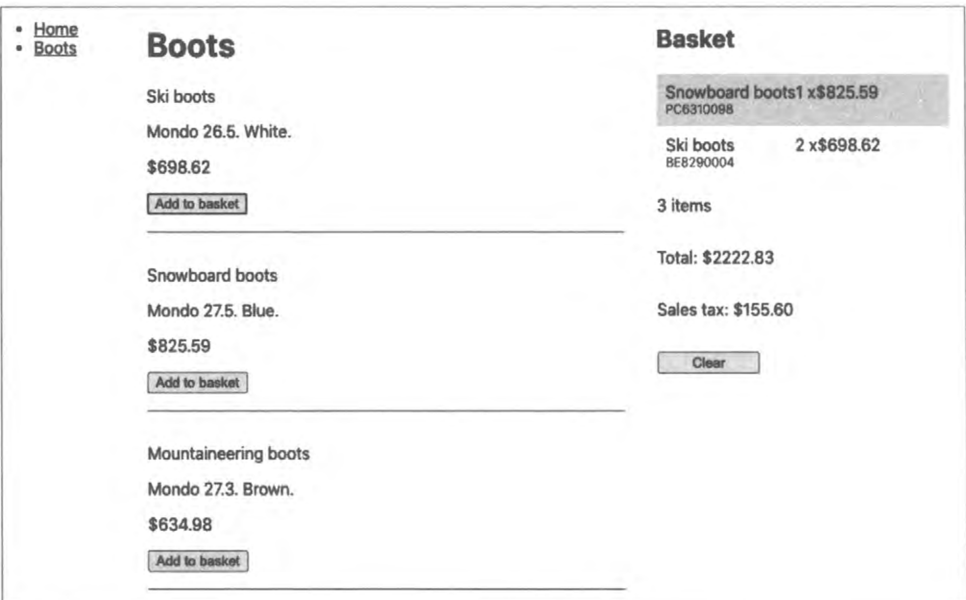


Рис. 3.22. Селекторные функции пересчитывают общую стоимость товаров в корзине и сумму налога только при изменении количества товаров в корзине

вполне можно использовать совместно с функциями-преобразователями, не относящимися к Redux. Поскольку у них нет зависимостей вне пределов самой библиотеки Reselect, они легко поддаются модульному тестированию. Примеры тестов включены в код этой главы.

Исходный код данного рецепта можно загрузить на веб-сайте GitHub по адресу **https://oreil.ly/U7SLr**.

Проектирование для обеспечения интерактивности

Рассматриваемые в этой главе рецепты фокусируются на решении целого ряда задач проектирования пользовательского интерфейса. В частности, обработка ошибок, предоставление помощи пользователям в применении разрабатываемой вами системы, создание сложных входных последовательностей, избегая при этом написания спагетти-кода и т. п.

Далее приведено множество советов, которые мы неоднократно находили полезными. В конце главы исследуются разные способы добавления анимации в приложения. Всюду, где это возможно, мы предпринимаем низкотехнологичный подход, и рассматриваемые здесь рецепты в идеале повысят эффективность ваших разработок интерфейса, требуя для этого минимальных усилий.

4.1. Создание центрального обработчика ошибок

ЗАДАЧА

Дать точное определение, что именно делает программное обеспечение хорошим, — задача не из легких. Но большинство превосходных программ имеют один общий аспект: их подход к реагированию на ошибки и исключения. При исполнении ваших приложений всегда будут возникать исключительные, неожиданные ситуации, например, потеря подключения к сети, сбой на сервере, искажение данных в хранилище и т. п. Поэтому важно рассмотреть, каким образом действовать при возникновении таких ситуаций.

Один из вариантов, который почти наверняка будет провальным, — это игнорировать факт возникновения сбойных ситуаций и прятать вызывающие их непривлекательные подробности. Но правильным подходом будет каким-либо образом организовать сохранение где-то подробностей обстоятельств, вызвавших сбой, которые затем можно будет использовать для предотвращения возникновения этой ошибки в будущем.

При разработке серверного кода подробности ошибки можно записывать в журнал и по запросу возвращать соответствующее сообщение. Но при написании клиентского кода требуется план, как действовать в случае возникновения локальных ошибок. Можно, например, отображать подробности ошибки пользователю и предложить отправить отчет об ошибке. Или можно задействовать стороннюю службу,

например Sentry.io (**https://sentry.io**), чтобы сохранять подробности ошибки удаленно.

Но какой бы подход ни был избран, он должен быть единообразен. Как же добиться единообразной обработки ошибок в приложениях React?

РЕШЕНИЕ

В этом рецепте мы рассмотрим, как создать центральный обработчик ошибок. Сразу же уточним, этот код не будет автоматически улавливать все исключения. Его еще нужно явно добавить в блоки перехвата ошибок (catch block) JavaScript. Это также не заменитель для обработки ошибок, после которых можно восстановить исполнение приложения каким-либо иным образом. Например, если при обработке заказа возникает ошибка по причине снятия сервера с линии для регламентных работ, будет намного лучше предложить пользователю повторить попытку позже.

Но этот подход поможет выявить любые ошибки, которые не были предусмотрены в нашем плане обработки ошибок.

Как правило, когда что-то идет не так, пользователю нужно сообщить следующие три вещи:

- Что случилось?
- Почему это случилось?
- Как реагировать на произошедшее?

В приводимом здесь примере мы будем обрабатывать ошибки, отображая диалоговое окно, содержащее подробности объекта `Error` JavaScript и предлагающее пользователю отправить электронной почтой его содержимое команде поддержки системы. При возникновении ошибки мы хотим вызывать простую функцию для ее обработки:

```
setVisibleError('Этого делать нельзя', errorObject)
```

Обычно обеспечить доступ к этой функции глобально по всему приложению можно посредством контекста. Контекст представляет собой нечто наподобие области видимости, в которую можно заключить набор компонентов React. Все, помещенное в этот контекст, становится доступным для всех дочерних элементов. В таком контексте мы сохраним нашу функцию обработки ошибок, которую сможем исполнять всякий раз, когда возникает ошибка.

Назовем наш контекст `ErrorHandlerContext` (листинг 4.1).

Листинг 4.1. Контекст для хранения функции обработки ошибок

```
import React from 'react'
const ErrorHandlerContext = React.createContext(() => {})
export default ErrorHandlerContext
```

Чтобы сделать контекст доступным набору компонентов, создадим компонент `ErrorHandlerProvider`, который, в свою очередь, создаст экземпляр контекста и сделает его доступным для всех передаваемых ему дочерних компонентов (листинг 4.2).

Листинг 4.2. Компонент ErrorHandlerProvider

```
import ErrorHandlerContext from './ErrorHandlerContext'
let setError = () => {}
const ErrorHandlerProvider = (props) => {
   if (props.callback) {
      setError = props.callback
   }
return (
   <ErrorHandlerContext.Provider value={setError}>
      {props.children}
   </ErrorHandlerContext.Provider>
   )
}
 export default ErrorHandlerProvider
```

Теперь нам нужно написать немного кода с инструкциями, что делать при вызове функции обработки ошибок. В данном случае такой код должен реагировать на сообщение об ошибке, отображая диалоговое окно, содержащее все подробности об ошибке. Соответствующий код приведен в листинге 4.3. Если вы хотите обрабатывать ошибки по-другому, модифицируйте этот код должным образом.

Листинг 4.3. Код, исполняющийся при сообщении об ошибке

```
import { useCallback, useState } from 'react'
import ErrorHandlerProvider from './ErrorHandlerProvider'
import ErrorDialog from './ErrorDialog'
const ErrorContainer = (props) => {
   const [error, setError] = useState()
   const [errorTitle, setErrorTitle] = useState()
   const [action, setAction] = useState()
   if (error) {
      console.error(
         'An error has been thrown',
         errorTitle,
         JSON.stringify(error)
      )
   }
   const callback = useCallback((title, err, action) => {
      console.error('ERROR RAISED ')
      console.error('Error title: ', title)
      console.error('Error content', JSON.stringify(err))
      setError(err)
      setErrorTitle(title)
      setAction(action)
   }, [])
```

```
  return (
    <ErrorHandlerProvider callback={callback}>
      {props.children}
      {error && (
        <ErrorDialog
          title={errorTitle}
          onClose={() => {
            setError(null)
            setErrorTitle('')
          }}
          action={action}
          error={error}
        />
      )}
    </ErrorHandlerProvider>
  )
}
export default ErrorContainer
```

Компонент `ErrorContrainer` отображает подробности ошибки посредством компонента `ErrorDialog`. Здесь мы не будем вдаваться в подробности кода компонента `ErrorDialog`, поскольку этот фрагмент нужно написать самостоятельно при реализации своей версии кода обработки ошибок[1].

В компонент `ErrorContainer` необходимо поместить большую часть нашего приложения. Вызывать обработчик ошибок смогут все компоненты внутри компонента `ErrorContainer` (листинг 4.4).

Листинг 4.4. Помещение приложения в компонент ErrorContainer

```
import './App.css'
import ErrorContainer from './ErrorContainer'
import ClockIn from './ClockIn'
function App() {
  return (
    <div className="App">
      <ErrorContainer>
        <ClockIn />
      </ErrorContainer>
    </div>
  )
}
  export default App
```

[1] Весь исходный код для данного рецепта можно загрузить на веб-сайте GitHub по адресу **https://oreil.ly/wUM7Q**.

Компонент использует обработчик ошибок посредством хука `useErrorHandler()` (листинг 4.5), который извлекает функцию обработки ошибок из контекста и возвращает ее.

Листинг 4.5. Создание хука `useErrorHandler`

```
import ErrorHandlerContext from './ErrorHandlerContext'
import { useContext } from 'react'
const useErrorHandler = () => useContext(ErrorHandlerContext)
export default useErrorHandler
```

Это был довольно сложный код, но теперь мы подошли к собственно использованию обработчика ошибок. И здесь уже нет ничего сложного. Например, код, приведенный в листинге 4.6, отправляет сетевой запрос, когда пользователь нажимает кнопку. Если исполнение запроса не завершается успешно, подробности возникшей ошибки передаются обработчику ошибок.

Листинг 4.6. Пример использования обработчика ошибок в коде

```
import useErrorHandler from './useErrorHandler'
import axios from 'axios'

const ClockIn = () => {
   const setVisibleError = useErrorHandler()

   const doClockIn = async () => {
      try {
         await axios.put('/clockTime')
      } catch (err) {
         setVisibleError('Unable to record work start time', err)
      }
   }
   return (
      <>
         <h1>Click Button to Record Start Time</h1>
         <button onClick={doClockIn}>Start work</button>
      </>
   )
}
export default ClockIn
```

На рис. 4.1 показано окно приложения.

Нажатие кнопки вызывает неуспешное завершение исполнения сетевого запроса из-за отсутствия серверного кода. Срабатывает обработчик ошибок и выводится диалоговое окно с описанием ошибки (рис. 4.2). Обратите внимание, что данное окно содержит информацию о том, какая произошла ошибка, почему она произошла и что пользователю следует предпринять относительно нее.

Рис. 4.1. Приложение записи времени начала работы

Unable to record work start time

Something bad happened. The details of the error are below. Please copy them and send them to systems support.


```
{
  "message": "Request failed with status code 404",
  "name": "Error",
  "stack": "Error: Request failed with status code 404\n    at
createError
(http://localhost:3000/static/js/vendors~main.chunk.js:10902:15)\n
at settle
(http://localhost:3000/static/js/vendors~main.chunk.js:11136:12)\n
at XMLHttpRequest.handleLoad
(http://localhost:3000/static/js/vendors~main.chunk.js:10376:7)\n
at XMLHttpRequest.eval (eval at makeContentWindowListener
(http://localhost:3000/__cypress/runner/cypress_runner.js:154555:10)
, <anonymous>:4:29)",
  "config": {
    "url": "/clockTime",
    "method": "put",
    "headers": {
      "Accept": "application/json, text/plain, */*"
    },
    "transformRequest": [
      null
```


COPY ERROR

Рис. 4.2. Когда сетевой запрос вызывает исключение, оно передается обработчику ошибок

Обсуждение

Изо всех рецептов, созданных нами в течение многих лет, описанный здесь сэкономил нам больше всего времени. В процессе разработки в исполнении кода часто возникают сбои, и если единственным указателем на причину ошибки является трассировка стека, упрятанная глубоко в консоли JavaScript, то его очень легко и не заметить.

Особенно важно, что в случае сбоя какой-либо части инфраструктуры (сети, шлюза, сервера, базы данных), этот небольшой фрагмент кода может сэкономить вам огромное количество часов, которые потребовалось бы потратить на поиски причины ошибки.

Исходный код данного рецепта можно загрузить на веб-сайте GitHub по адресу **https://oreil.ly/wUM7Q**.

4.2. Создаем интерактивное справочное руководство

ЗАДАЧА

Тим Бернерс-Ли (Tim Berners-Lee) преднамеренно оснастил Всемирную паутину очень небольшим количеством возможностей. Она работает по простому протоколу (HTTP), и первоначально в ней использовался простой язык разметки (HTML). Такое отсутствие сложности означало, что новые пользователи веб-сайтов сразу же понимали, как работать с ними. Например, если что-то выглядело как гиперссылка, было ясно, что щелчок по ней переведет тебя на другую страницу.

Но все это изменилось с приходом насыщенных приложений JavaScript. Веб-приложения больше не являются набором веб-страниц, связанных между собой гиперссылками. Теперь они больше похожи на старые компьютерные приложения, более мощные и с большим количеством возможностей. Но за это приходится расплачиваться намного большей сложностью работы с ними, зачастую требуя специального руководства, объясняющего, что и как.

Итак, как можно добавить интерактивное руководство в наше приложение?

РЕШЕНИЕ

Мы создадим простую справочную систему, которую можно будет наложить поверх существующего приложения. При открытии страницы справки отображается последовательность всплывающих окон, описывающих, как работать с приложением (рис. 4.3).

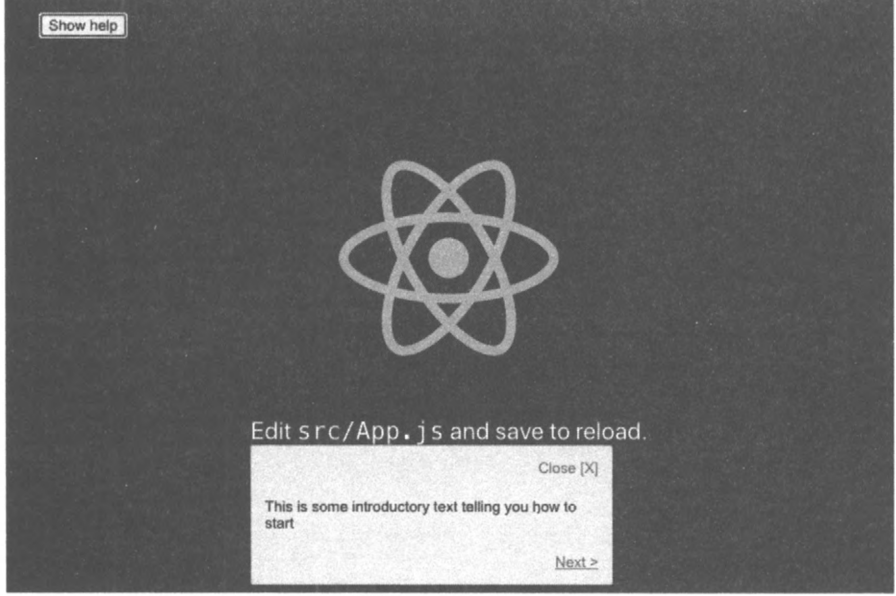


Рис. 4.3. При нажатии кнопки **Show help** выводится первое из последовательности всплывающих окон с инструкциями по работе с приложением

Мы хотим, чтобы нашу справочную систему было легко обслуживать и чтобы она предоставляла информацию только по видимым компонентам страницы. Поскольку это довольно объемная задача, начнем с создания компонента, который выводит всплывающее окно со справочной информацией (листинг 4.7).

Листинг 4.7. Компонент для отображения всплывающего окна с подсказками

```
import { Popper } from '@material-ui/core'
import './HelpBubble.css'

const HelpBubble = (props) => {
  const element = props.forElement
    ? document.querySelector(props.forElement)
    : null
  return element ? (
    <Popper
      className="HelpBubble-container"
      open={props.open}
      anchorEl={element}
      placement={props.placement || 'bottom-start'}
    >
      <div className="HelpBubble-close" onClick={props.onClose}>
        Close [X]
      </div>
      {props.content}
      <div className="HelpBubble-controls">
        {props.previousLabel ? (
          <div
            className="HelpBubble-control HelpBubble-previous"
            onClick={props.onPrevious}
          >
            &lt; {props.previousLabel}
          </div>
        ) : (
          <div> </div>
        )}
        {props.nextLabel ? (
          <div
            className="HelpBubble-control HelpBubble-next"
            onClick={props.onNext}
          >
            {props.nextLabel} &gt;
          </div>
        ) : (
          <div> </div>
        )}
```

```
      </div>
    </Popper>
  ) : null
}
export default HelpBubble
```

В листинге 4.7 используется компонент `Popper` из библиотеки @material-ui. Этот компонент можно закрепить на странице рядом с каким-либо другим компонентом. Компонент `HelpBubble` принимает строку `forElement`, которая будет представлять какой-либо селектор CSS, например `.class-name` или `#some-id`. Селекторы будут служить для ассоциирования элементов на экране со всплывающими сообщениями.

Теперь, когда у нас есть компонент для всплывающих сообщений, нам нужен компонент для координации отображения таких всплывающих сообщений. Код такого компонента, называющегося `HelpSequence`, приведен в листинге 4.8.

Листинг 4.8. Код компонента `HelpSequence`

```
import { useEffect, useState } from 'react'
import HelpBubble from './HelpBubble'
function isVisible(e) {
  return !!(
    e.offsetWidth ||
    e.offsetHeight ||
    e.getClientRects().length
  )
}
const HelpSequence = (props) => {
  const [position, setPosition] = useState(0)
  const [sequence, setSequence] = useState()
useEffect(() => {
  if (props.sequence) {
    const filter = props.sequence.filter((i) => {
      if (!i.forElement) {
        return false
      }
      const element = document.querySelector(i.forElement)
      if (!element) {
        return false
      }
      return isVisible(element)
    })
    setSequence(filter)
  } else {
    setSequence(null)
  }
}, [props.sequence, props.open])
```

```
const data = sequence && sequence[position]
useEffect(() => {
   setPosition(0)
}, [props.open])
const onNext = () =>
   setPosition((p) => {
      if (p === sequence.length - 1) {
         props.onClose && props.onClose()
      }
      return p + 1
      })
   const onPrevious = () =>
      setPosition((p) => {
         if (p === 0) {
            props.onClose && props.onClose()
         }
         return p - 1
      })
   return (
      <div className="HelpSequence-container">
         {data && (
            <HelpBubble
               open={props.open}
               forElement={data.forElement}
               placement={data.placement}
               onClose={props.onClose}
               previousLabel={position > 0 && 'Previous'}
               nextLabel={
                  position < sequence.length - 1 ? 'Next' : 'Finish'
               }
               onPrevious={onPrevious}
               onNext={onNext}
               content={data.text}
            />
         )}
      </div>
   )
}
export default HelpSequence
```

Компонент `HelpSequence` принимает массив объектов JavaScript, пример которого показан в листинге 4.9.

Листинг 4.9. Пример объектов JavaScript, принимаемых компонентом `HelpSequence`

```
[
   {forElement: "p",
      text: "This is some introductory text telling you how to start"},
```

```
  {forElement: ".App-link", text: "This will show you how to use React"},
  {forElement: ".App-nowhere", text: "This help text will never appear"},
]
```

В компоненте `HelpSequence` эти объекты преобразуются в динамическую последовательность компонентов `HelpBubbles`. Компоненты `HelpBubbles` отображаются только при обнаружении элемента, соответствующего селектору `forElement`. В таком случае возле этого элемента отображается всплывающее окно `HelpBubble`, содержащее текст соответствующей справки.

Добавим теперь компонент `HelpSequence` в стандартный код файла App.js, созданный средством `create-react-app` (листинг 4.10).

Листинг 4.10. Вставка компонента `HelpSequence` в стандартный код App.js

```
import { useState } from 'react'
import logo from './logo.svg'
import HelpSequence from './HelpSequence'
import './App.css'
function App() {
  const [showHelp, setShowHelp] = useState(false)
  return (
    <div className="App">
      <header className="App-header">
        <img src={logo} className="App-logo" alt="logo" />
        <p>
          Edit <code>src/App.js</code> and save to reload.
        </p>
        <a
          className="App-link"
          href="https://reactjs.org"
          target="_blank"
          rel="noopener noreferrer"
        >
          Learn React
        </a>
      </header>
      <button onClick={() => setShowHelp(true)}>Show help</button>
      <HelpSequence
        sequence={[
          {
            forElement: 'p',
            text: 'This is some introductory text telling you how to start',
          },
          {
            forElement: '.App-link',
            text: 'This will show you how to use React',
          },
```

```
            {
               forElement: '.App-nowhere',
               text: 'This help text will never appear',
            },
         ]}
         open={showHelp}
         onClose={() => setShowHelp(false)}
      />
   </div>
 )
}
export default App
```

Сразу после запуска внешний вид модифицированного приложения практически ничем не отличается от обычного стандартного приложения, но добавлена кнопка **Show help** (рис. 4.4).

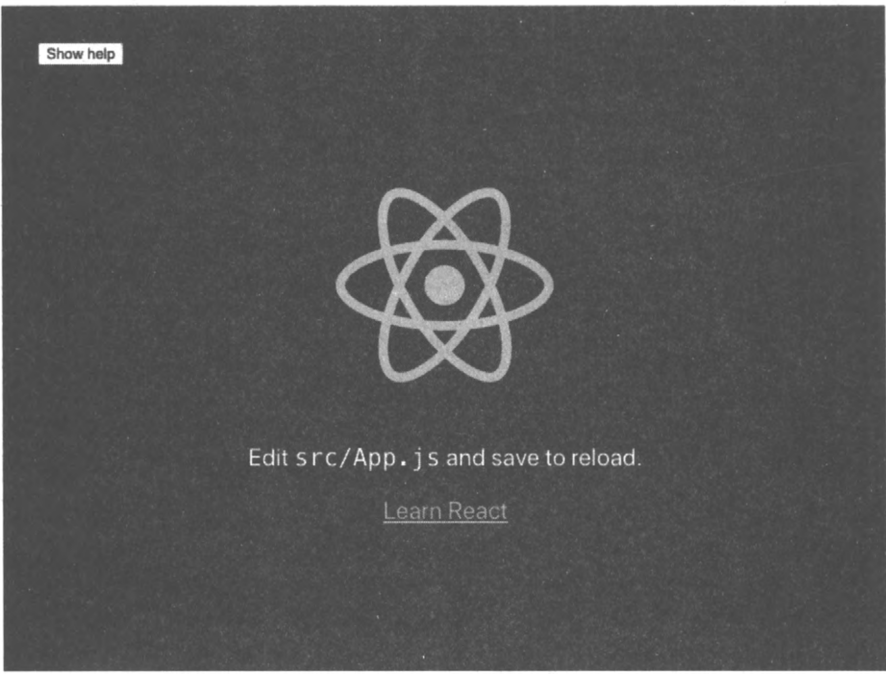


Рис. 4.4. Внешний вид приложения сразу после загрузки

При нажатии этой кнопки отображается диалоговое окно со справочной информацией для первого элемента, как показано на рис. 4.5.

А при нажатии в этом окне ссылки **Next** открывается справочное диалоговое окно для следующего элемента, как показано на рис. 4.6. Нажатие ссылки **Next** в каждом справочном диалоговом окне будет последовательно отображать справочное окно для следующего элемента, пока не останется совпадающих элементов.

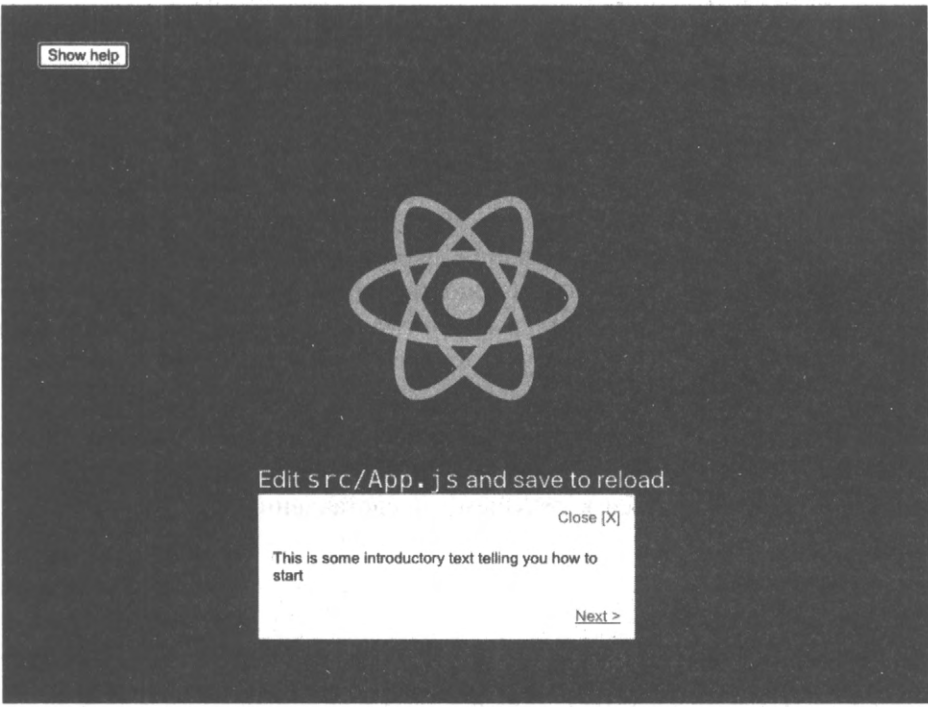


Рис. 4.5. При нажатии кнопки **Show help** открывается справочное диалоговое окно для первого элемента

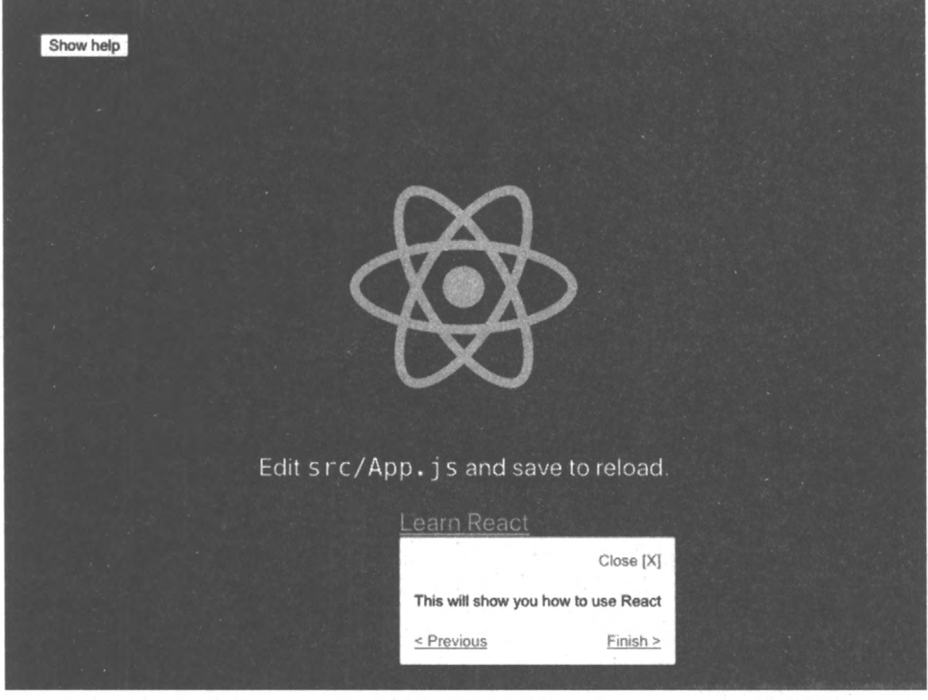


Рис. 4.6. В последнем справочном окне вместо ссылки **Next** используется ссылка **Finish**

Обсуждение

Присутствие в приложении интерактивной справки делает возможным для пользователя узнавать о наличии, назначении и способе использования элементов его пользовательского интерфейса. Разработчики уделяют много времени, добавляя в приложения возможности, которые, вероятно, никогда не будут задействованы просто по той причине, что пользователи не будут знать о них.

В нашей реализации справки информация предоставляется в виде простого текста. Но может быть полезным рассмотреть использование для этой цели языка разметки Markdown, чтобы обеспечить более насыщенную справочную среду, а также иметь возможность вставлять ссылки на другие, более обширные справочные страницы[2].

Справочные темы автоматически ограничиваются элементами, видимыми на странице. Можно создать или отдельную справочную последовательность для каждой страницы, или одну большую справочную последовательность, которая будет автоматически адаптироваться к текущему представлению пользовательского интерфейса.

Наконец, справочная система наподобие рассмотренной здесь идеально подходит для хранения в "безголовых" системах CMS, что позволит обновлять справку динамически, без необходимости создавать новое развертывание при каждом обновлении.

Исходный код данного рецепта можно загрузить на веб-сайте GitHub по адресу **https://oreil.ly/CsiMN**.

4.3. Сложные взаимодействия посредством преобразователей

ЗАДАЧА

Часто для получения желаемого результата работы приложения пользователям нужно выполнить определенную последовательность действий. Например, проделать последовательность шагов в каком-либо мастере или осуществить вход в систему и подтвердить намерение выполнить какую-либо опасную операцию (рис. 4.7).

Кроме того, пользователю может потребоваться не только проделать последовательность шагов, но и выполнение самих шагов может быть обусловленным. Например, если пользователь недавно вошел в систему, то повторный вход может не требоваться. Или же пользователь может решить отменить операцию посередине последовательности шагов.

При моделировании сложных последовательностей в компонентах можно вскоре обнаружить, что приложение битком набито спагетти-кодом.

---

[2] Дополнительная информация по использованию редактора Markdown в приложениях приведена в *разделе 4.5*.

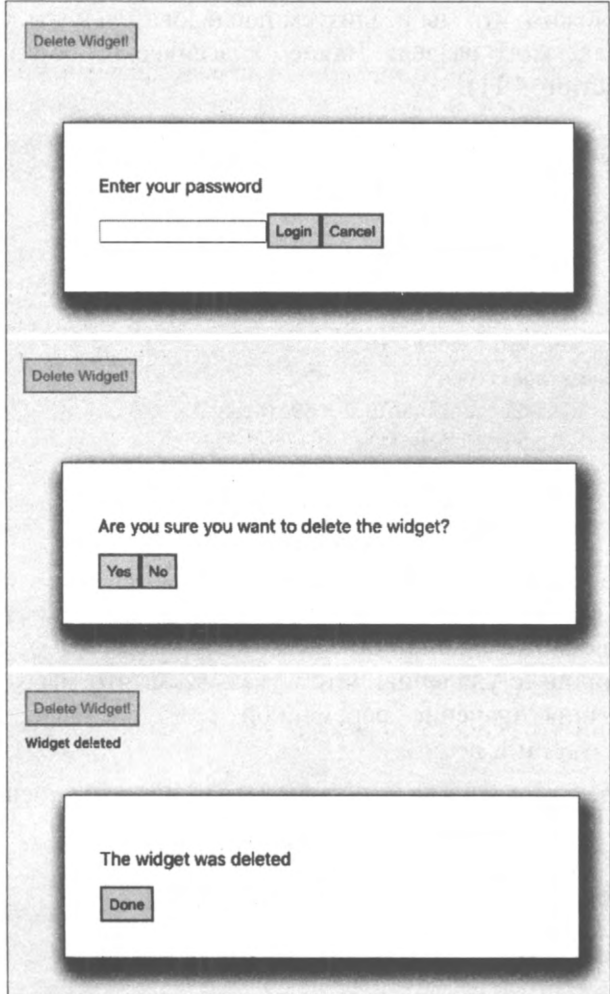


Рис. 4.7. Процесс удаления виджета требует входа в систему по паролю и последующего подтверждения удаления

РЕШЕНИЕ

Для управления сложной последовательностью операций мы рекомендуем прибегнуть к преобразователю. Мы познакомились с использованием преобразователей для управления состоянием в *главе 3*. Преобразователь — это функция JavaScript, которая принимает в качестве параметров объект состояния и объект действия. На основе действия принимается решение, как изменить состояние, и оно не должно иметь побочных эффектов.

Поскольку преобразователи не имеют кода пользовательского интерфейса, они идеально подходят для управления запутанными частями взаимосвязанного состояния, без необходимости заботиться о внешнем виде. Они особенно хорошо поддаются модульному тестированию.

Например, предположим, что мы реализуем последовательность удаления виджета, упомянутую в начале этого раздела. Начнем классическим образом с создания модульного теста (листинг 4.11).

Листинг 4.11. Код модульного теста приложения

```
import deletionReducer from './deletionReducer'
describe('deletionReducer', () => {
  it('should show the login dialog if we are not logged in', () => {
    const actual = deletionReducer({}, { type: 'START_DELETION' })
    expect(actual.showLogin).toBe(true)
    expect(actual.message).toBe('')
    expect(actual.deleteButtonDisabled).toBe(true)
    expect(actual.loginError).toBe('')
    expect(actual.showConfirmation).toBe(false)
  })
})
```

Здесь мы создали функцию преобразования, называющуюся `deletionReducer`. Передаем ей пустой объект (`{}`) и действие (`{type: 'START_DELETION'}`), указывающее, что мы хотим начать процесс удаления. Затем указываем, что мы хотим, чтобы в новой версии состояния значение переменной `showLogin` было `true`, переменной `showConfirmation` — `false` и т. д.

Затем мы можем реализовать код преобразователя для выполнения этих операций (листинг 4.12).

Листинг 4.12. Код функции преобразования

```
function deletionReducer(state, action) {
  switch (action.type) {
    case 'START_DELETION':
      return {
        ...state,
        showLogin: true,
        message: '',
        deleteButtonDisabled: true,
        loginError: '',
        showConfirmation: false,
      }
    default:
      return null // Or anything
    }
  }
```

Сначала мы просто присваиваем атрибутам состояния значения, которые успешно проходят тестирование. По мере добавления дополнительных тестов качество на-

шего преобразователя повышается, т. к. он может обрабатывать большее количество ситуаций.

В конечном итоге наш преобразователь будет выглядеть так, как показано в листинге 4.13[3].

Листинг 4.13. Конечная версия функции преобразователя

```
function deletionReducer(state, action) {
   switch (action.type) {
      case 'START_DELETION':
         return {
            ...state,
            showLogin: !state.loggedIn,
            message: '',
            deleteButtonDisabled: true,
            loginError: '',
            showConfirmation: !!state.loggedIn,
         }
      case 'CANCEL_DELETION':
         return {
            ...state,
            showLogin: false,
            showConfirmation: false,
            showResult: false,
            message: 'Deletion canceled',
            deleteButtonDisabled: false,
         }
      case 'LOGIN':
         const passwordCorrect = action.payload === 'swordfish'
         return {
            ...state,
            showLogin: !passwordCorrect,
            showConfirmation: passwordCorrect,
            loginError: passwordCorrect ? '' : 'Invalid password',
            loggedIn: true,
         }
      case 'CONFIRM_DELETION':
         return {
            ...state,
            showConfirmation: false,
            showResult: true,
            message: 'Widget deleted',
         }
```

[3] Тесты для проверки работы этого кода можно загрузить из репозитория GitHub по адресу **https://oreil.ly/DCGIv**.

```
      case 'FINISH':
         return {
            ...state,
            showLogin: false,
            showConfirmation: false,
            showResult: false,
            deleteButtonDisabled: false,
         }
      default:
         throw new Error('Unknown action: ' + action.type)
   }
}
export default deletionReducer
```

Несмотря на сложность приведенного кода, его можно написать довольно быстро, если начать с разработки тестов.

Имея в своем распоряжении преобразователь, используем его в нашем приложении (листинг 4.14).

Листинг 4.14. Применение функции преобразователя в приложении

```
import { useReducer, useState } from 'react'
import './App.css'
import deletionReducer from './deletionReducer'
function App() {
   const [state, dispatch] = useReducer(deletionReducer, {})
   const [password, setPassword] = useState()
   return (
      <div className="App">
         <button
            onClick={() => {
               dispatch({ type: 'START_DELETION' })
            }}
            disabled={state.deleteButtonDisabled}
         >
            Delete Widget!
         </button>
         <div className="App-message">{state.message}</div>
         {state.showLogin && (
            <div className="App-dialog">
               <p>Enter your password</p>
                  <input
                     type="password"
                     value={password}
                     onChange={(evt) => setPassword(evt.target.value)}
                  />
```

```
        <button
          onClick={() =>
            dispatch({ type: 'LOGIN', payload: password })
          }
        >
          Login
        </button>
        <button
          onClick={() => dispatch({ type: 'CANCEL_DELETION' })}
        >
          Cancel
        </button>
          <div className="App-error">{state.loginError}</div>
        </div>
      )}
      {state.showConfirmation && (
        <div className="App-dialog">
          <p>Are you sure you want to delete the widget?</p>
          <button
            onClick={() =>
              dispatch({
                type: 'CONFIRM_DELETION',
              })
            }
          >
            Yes
          </button>
          <button
            onClick={() =>
              dispatch({
                type: 'CANCEL_DELETION',
              })
            }
          >
            No
          </button>
        </div>
      )}
      {state.showResult && (
        <div className="App-dialog">
          <p>The widget was deleted</p>
          <button
            onClick={() =>
              dispatch({
                type: 'FINISH',
              })
            }
          >
```

```
                        Done
                    </button>
                </div>
            )}
        </div>
    )
}
export default App
```

Бóльшая часть кода этого компонента просто создает пользовательский интерфейс для каждого диалогового окна последовательности и не содержит практически никакой логики. Просто реализуется то, что указывает преобразователь. В итоге пользователь получит так называемый *беспроблемный путь* (happy path) входа в систему и подтверждения удаления виджета (рис. 4.8).

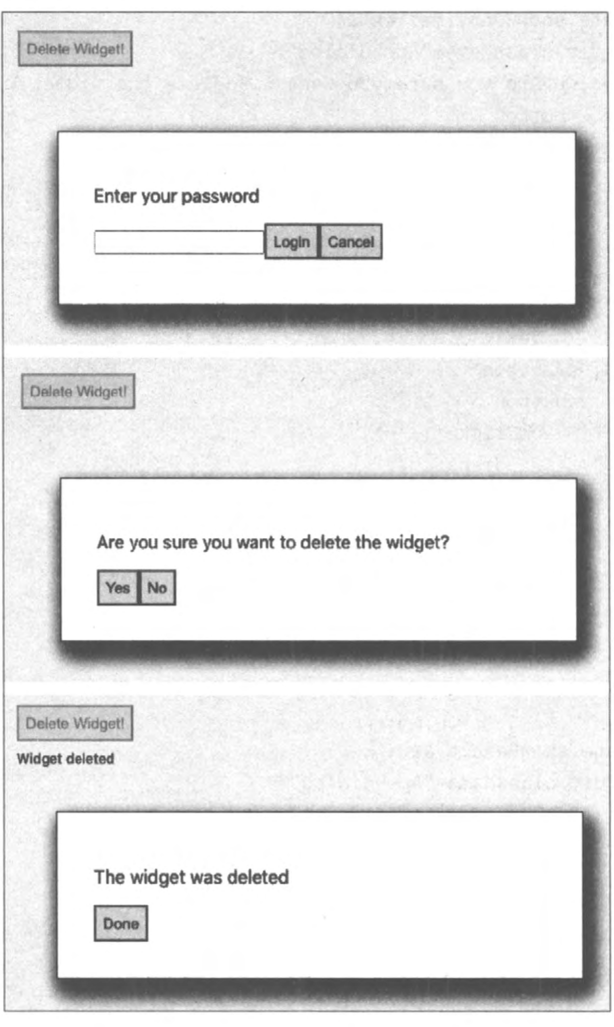


Рис. 4.8. Конечный результат исполнения приложения

Но, как можно видеть из рис. 4.9, программа также обрабатывает и случаи ухода с обычного пути (edge case), такие как ввод неправильного пароля и отмена операции удаления.

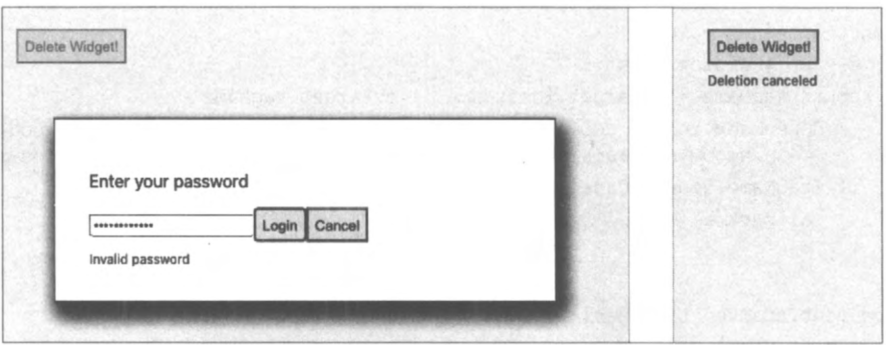


Рис. 4.9. Обработка нештатных ситуаций преобразователем

Обсуждение

В некоторых случаях наличие преобразователей может сделать код запутанным. В ситуациях, когда нужно обрабатывать только малое число состояний с небольшим количеством взаимодействий между ними, преобразователь, скорее всего, не требуется. Но если для описания последовательности взаимодействий пользователя не обойтись без блок-схемы или диаграммы состояний, значит, может потребоваться прибегнуть к помощи преобразователя.

Исходный код данного рецепта можно загрузить на веб-сайте GitHub по адресу **https://oreil.ly/hfqLn**.

4.4. Взаимодействие с клавиатурой

ЗАДАЧА

Опытные пользователи предпочитают исполнять часто встречающиеся операции при помощи клавиатуры. Компоненты React могут реагировать на события клавиатуры, но только тогда, когда они (или их потомки) имеют фокус. Каким образом можно реализовать реагирование компонента на события на уровне документа?

РЕШЕНИЕ

Для решения этой задачи мы создадим хук, который будет отслеживать события нажатия клавиши клавиатуры `keydown` на уровне документа. При этом его можно легко переделать для отслеживания любого другого события JavaScript в иерархии DOM. Код этого хука приведен в листинге 4.15.

Листинг 4.15. Хук для отслеживания событий клавиатуры keydown

```
import { useEffect } from 'react'
const useKeyListener = (callback) => {
   useEffect(() => {
      const listener = (e) => {
         e = e || window.event
         const tagName = e.target.localName || e.target.tagName
         // Принимаем только события клавиш, возникающие на уровне элемента body,
               чтобы избежать включения нажатий клавиш, например, в текстовых полях
         if (tagName.toUpperCase() === 'BODY') {
            callback(e)
         }
      }
      document.addEventListener('keydown', listener, true)
      return () => {
         document.removeEventListener('keydown', listener, true)
      }
   }, [callback])
}
export default useKeyListener
```

Хук принимает функцию обратного вызова и регистрирует ее для событий нажатия клавиш `keydown` по объекту `document`. В конце `useEffect` он возвращает функцию, которая отменяет регистрацию функции обратного вызова. При изменении передаваемой функции обратного вызова, прежде чем регистрировать новую функцию, выполняется отмена регистрации предыдущей.

В листинге 4.16 приведен пример использования нашего хука. Подумайте, все ли здесь правильно, сможете ли вы обнаружить небольшую проблему с написанием данного кода, которую нужно было решить.

Листинг 4.16. Пример использования хука для отслеживания событий keydown

```
import { useCallback, useState } from 'react'
import './App.css'
import useKeyListener from './useKeyListener'
const RIGHT_ARROW = 39
const LEFT_ARROW = 37
const ESCAPE = 27
function App() {
   const [angle, setAngle] = useState(0)
   const [lastKey, setLastKey] = useState('')
   let onKeyDown = useCallback(
      (evt) => {
         if (evt.keyCode === LEFT_ARROW) {
            setAngle((c) => Math.max(-360, c - 10))
            setLastKey('Left')
```

```
        } else if (evt.keyCode === RIGHT_ARROW) {
           setAngle((c) => Math.min(360, c + 10))
           setLastKey('Right')
        } else if (evt.keyCode === ESCAPE) {
           setAngle(0)
           setLastKey('Escape')
        }
     },
     [setAngle]
  )
  useKeyListener(onKeyDown)
     return (
        <div className="App">
        <p>
           Angle: {angle} Last key: {lastKey}
        </p>
        <svg
           width="400px"
           height="400px"
           title="arrow"
           fill="none"
           strokeWidth="10"
           stroke="black"
           style={{
              transform: `rotate(${angle}deg)`,
           }}
        >
           <polyline points="100,200 200,0 300,200" />
           <polyline points="200,0 200,400" />
        </svg>
     </div>
  )
}
export default App
```

Программа ожидает нажатия пользователем клавиши правой или левой стрелки. Обработка этих событий осуществляется функцией `onKeyDown`, но обратите внимание на то, что мы заключили ее в `useCallback`. Если этого не сделать, то браузер будет воссоздавать функцию `onKeyDown` всякий раз при отрисовке компонента `App`. Новая функция делает то же самое, что и старая, но она будет находиться в другом месте в памяти, и `useKeyListener` будет постоянно отменять ее регистрацию и повторно регистрировать ее.

Если функцию обратного вызова не заключить в `useCallback`, это может вызвать поток вызовов для отрисовки, что может замедлить работу приложения.

Благодаря `useCallback` мы можем обеспечить создание функции только при изменении `setAngle`.

При запуске приложения на исполнение на экран выводится стрелка, которую можно вращать, нажимая клавиши левой и правой стрелки на клавиатуре (рис. 4.10). Нажатие клавиши <Escape> возвращает стрелку в исходное вертикальное положение.

Рис. 4.10. Управление стрелкой на экране посредством нажатия клавиш клавиатуры

Обсуждение

Мы предприняли должные меры предосторожности, чтобы функция `useKeyListener` отслеживала только события нажатия клавиш, возникающие лишь на уровне элемента `body`. Браузер не передает программе события нажатия клавиш стрелок, происходящие, когда фокус находится в текстовом поле.

Исходный код данного рецепта можно загрузить на веб-сайте GitHub по адресу **https://oreil.ly/VIY1O**.

4.5. Создание насыщенного содержимого посредством редактора Markdown

ЗАДАЧА

В приложениях, предоставляющих пользователям возможность работы с большими блоками текста, полезно сделать так, чтобы этот текст содержал должное форматирование, ссылки и т. п. Но если передавать пользователю такие "ужасы", как необработанный код HTML, то могут возникнуть проблемы безопасности и несказанные страдания разработчиков.

Как же обеспечить пользователям ввод высокоинформативного содержимого, не подрывая безопасность приложения?

РЕШЕНИЕ

Задача предоставления пользователям возможности безопасно передавать в приложение объемное содержимое прекрасно решается посредством редактора Mark-

down. Чтобы увидеть, как работать с редактором Markdown, рассмотрим следующее простое приложение (листинг 4.17), которое позволяет пользователю вставлять в список ряд сообщений с отметкой времени.

Листинг 4.17. Приложение для ввода простых текстовых сообщений с отметкой времени

```
import { useState } from 'react'
import './Forum.css'
   const Forum = () => {
   const [text, setText] = useState('')
   const [messages, setMessages] = useState([])
return (
   <section className="Forum">
      <textarea
      cols={80}
      rows={20}
      value={text}
      onChange={(evt) => setText(evt.target.value)}
   />
   <button
      onClick={() => {
         setMessages((msgs) => [
            {
               body: text,
               timestamp: new Date().toISOString(),
            },
            ...msgs,
         ])
         setText('')
      }}
   >
      Post
   </button>
   {messages.map((msg) => {
      return (
            <dl>
               <dt>{msg.timestamp}</dt>
               <dd>{msg.body}</dd>
            </dl>
            )
         })}
      </section>
   )
}
export default Forum
```

При исполнении приложения отображается большое текстовое поле (рис. 4.11). Когда пользователь вводит в поле простой текст, приложение сохраняет пробелы и переходы на новую строку.

```
# Talk outline

The following is the *draft* outline for today's talk.

* Introduction
* NLS as an instrument
* Control techniques
* NLS implementation
* Usage
* Activities
```

Post

Post

2021-05-15T22:49:12.472Z

```
# Talk outline

The following is the *draft* outline for today's talk.

* Introduction
* NLS as an instrument
* Control techniques
* NLS implementation
* Usage
* Activities
```

Рис. 4.11. Вводимый в текстовое поле текст отображается на странице приложения

Для приложений, использующих текстовые поля, целесообразно разрешить пользователю вводить содержимое с помощью редактора Markdown.

Имеется множество библиотек Markdown, но большинство из них представляют собой просто оболочки для компонента `react-markdown` или выделители синтаксических элементов типа PrismJS (**https://prismjs.com**) или CodeMirror (**https://codemirror.net**).

Здесь мы рассмотрим библиотеку `react-md-editor`, которая добавляет дополнительные возможности в компонент `react-markdown`, позволяя отображать поле редактора Markdown и редактировать текст в нем. Первым делом установим эту библиотеку:

```
$ npm install @uiw/react-md-editor
```

Теперь мы можем преобразовать поле ввода простого текста в редактор Markdown и форматировать вводимые сообщения языком HTML (листинг 4.18).

Листинг 4.18. Использование библиотеки react-md-editor

```
import { useState } from 'react'
import MDEditor from '@uiw/react-md-editor'
const MarkdownForum = () => {
  const [text, setText] = useState('')
  const [messages, setMessages] = useState([])
  return (
    <section className="Forum">
      <MDEditor height={300} value={text} onChange={setText} />
      <button
        onClick={() => {
          setMessages((msgs) => [
              {
                body: text,
                timestamp: new Date().toISOString(),
              },
              ...msgs,
            ])
            setText('')
        }}
      >
        Post
      </button>
      {messages.map((msg) => {
        return (
          <dl>
            <dt>{msg.timestamp}</dt>
            <dd>
              <MDEditor.Markdown source={msg.body} />
            </dd>
          </dl>
        )
      })}
    </section>
  )
}
export default MarkdownForum
```

Небольшие усилия по преобразованию простого текстового поля в поле с возможностями редактора Markdown дают значительную отдачу. Как можно видеть из рис. 4.12, пользователь может применять сложное форматирование (rich formatting) к вводимому тексту, а также имеет возможность редактировать его в полноэкранном режиме перед сохранением.

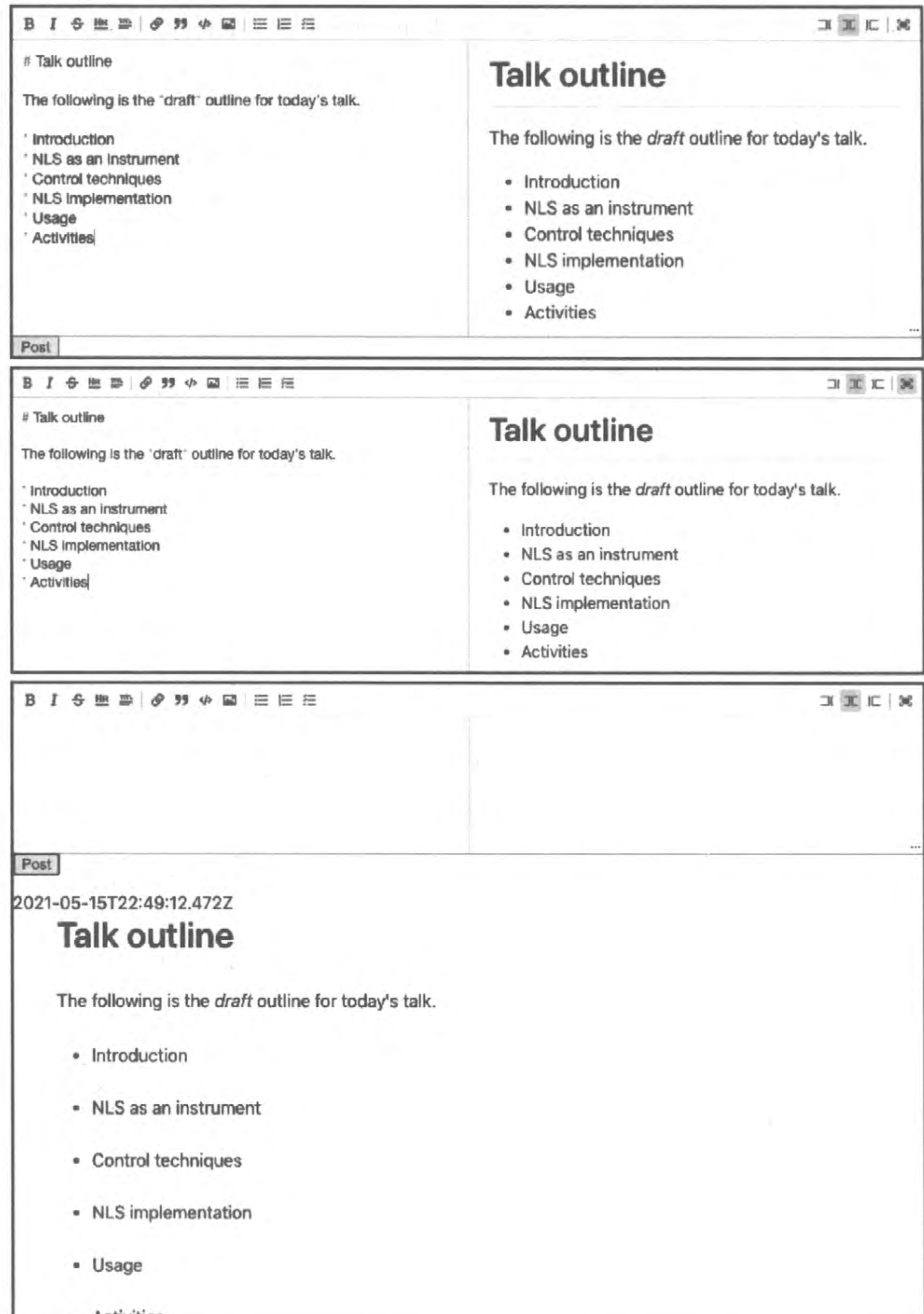


Рис. 4.12. Редактор Markdown обеспечивает предварительный просмотр вводимого текста и редактирование в полноэкранном режиме

Обсуждение

Добавление в приложение возможностей редактора Markdown не занимает много времени и требует минимальных усилий, но значительно улучшает восприятие приложения пользователем. Подробная информация по редактору Markdown предоставляется в оригинальном руководстве его разработчика Джона Грубера (John Gruber), доступном по адресу **https://oreil.ly/2EE9x**.

Исходный код данного рецепта можно загрузить на веб-сайте GitHub по адресу **https://oreil.ly/S0n7x**.

4.6. Анимация посредством классов CSS

ЗАДАЧА

В разрабатываемое приложение требуется добавить небольшую простую анимацию, не прибегая при этом к установке сторонних библиотек, чтобы не увеличивать размер приложения.

РЕШЕНИЕ

Для большинства анимационных эффектов, которые вам когда-либо потребуются в приложении React, установка сторонних библиотек, скорее всего, не будет обязательной. В настоящее время средства CSS дают браузерам нативную возможность анимировать свойства CSS с минимальными усилиями. Анимация получается плавная, не требуя большого объема кода, поскольку она генерируется аппаратными графическими средствами. Анимация посредством графического процессора менее энергозатратная, что делает ее применение более подходящим для мобильных устройств.

Если вы хотите добавить анимационные эффекты в свои приложения React, начните с возможностей CSS, прежде чем рассматривать другие подходы.

Анимация средствами CSS основана на свойстве CSS, называющемся *переходом* (transition). Предположим, что мы хотим создать расширяющуюся информационную панель. При нажатии пользователем кнопки на панели она должна раскрыться, а при повторном нажатии кнопки — так же плавно закрыться (рис. 4.13).

Такой анимационный эффект можно создать при помощи свойства `transition` CSS, показанного в листинге 4.19.

Листинг 4.19. Свойство `transition` CSS

```
.InfoPanel-details {
  height: 350px;
  transition: height 0.5s;
}
```

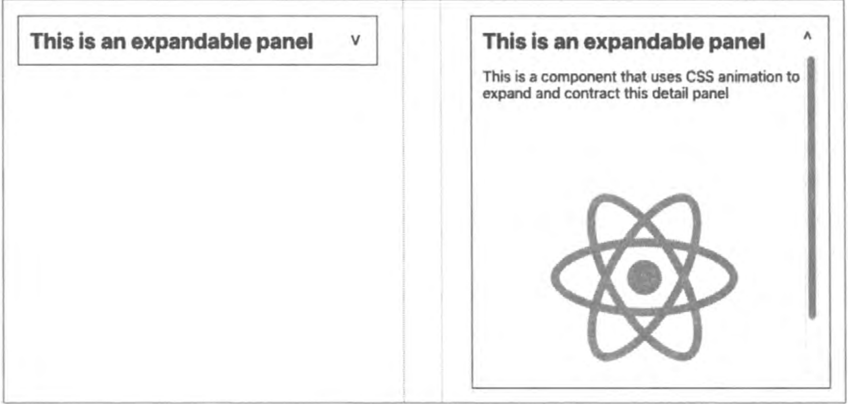


Рис. 4.13. Простая CSS-анимация плавно раскрывает и закрывает информационную панель

В данном правиле CSS указываются свойства `height` и `transition`. Эта комбинация задает плавное изменение текущей высоты (`height`) элемента к требуемой в течение следующей полсекунды.

Анимационный эффект запускается при изменении высоты элемента, например когда становится действительным другое правило CSS. Если имеется другое имя класса CSS, в котором указана иная высота (листинг 4.20), то свойство `transition` выполнит изменение высоты с анимационным эффектом при переключении элемента на другой класс.

Листинг 4.20. Определение другого имени класса CSS с другой высотой элемента

```
.InfoPanel-details {
  height: 350px;
  transition: height 0.5s;
}
.InfoPanel-details.InfoPanel-details-closed {
  height: 0;
}
```

Данная структура именования классов — это пример именования BEM (Block Element Modifier — модификатор блочных элементов). *Блоком* в данном случае является компонент (`InfoPanel`), *элементом* — содержимое блока (`details`), а *модификатор* сообщает текущее состояние элемента (`closed`). Схема именования BEM понижает вероятность конфликта имен в программе.

Если элемент `InfoPanels-details` внезапно получит дополнительный класс `.InfoPanel-details-closed`, то его высота изменится с `350px` до `0` и свойство `transition` плавно свернет элемент. И наоборот, при потере класса `.InfoPanel-details-closed` элемент снова будет развернут.

Все это означает, что мы можем оставить всю тяжелую работу CSS, а в коде React нам нужно будет только добавлять класс к элементу или удалять его, как показано в листинге 4.21.

Листинг 4.21. Манипулирование классами элемента в коде React

```
import { useState } from 'react'
import './InfoPanel.css'
const InfoPanel = ({ title, children }) => {
   const [open, setOpen] = useState(false)
   return (
      <section className="InfoPanel">
         <h1>
            {title}
           <button onClick={() => setOpen((v) => !v)}>
              {open ? '^' : 'v'}
            </button>
         </h1>
         <div
            className={`InfoPanel-details ${
               open ? '' : 'InfoPanel-details-closed'
            }`}
         >
            {children}
         </div>
      </section>
   )
}
export default InfoPanel
```

Обсуждение

Нам часто приходилось видеть включение в проекты сторонних библиотек компонентов для использования какого-либо небольшого виджета, который разворачивает или сворачивает свое содержимое. Как было продемонстрировано выше, добавление такого анимационного эффекта в приложение не представляет никаких сложностей.

Исходный код данного рецепта можно загрузить на веб-сайте GitHub по адресу **https://oreil.ly/FKnIc**.

4.7. Анимация средствами React

ЗАДАЧА

Хотя анимационные средства CSS очень низкотехнологичные, они будут достаточными для большинства задач анимации, которые вам, вероятно, могут потребоваться.

Но они требуют глубокого понимания различных свойств CSS и эффектов их анимации. Например, как можно проиллюстрировать удаление элемента, резко его расширяя и делая прозрачным?

Многие библиотеки, например библиотека Animate.css (**https://animate.style**), содержат большое множество готовых анимаций CSS, но для них часто требуется применение более продвинутых принципов CSS-анимации, таких как опорные кадры (keyframe). Кроме того, они не особенно заточены под React. Так как же нам добавить в приложение React анимации посредством CSS-библиотек?

РЕШЕНИЕ

Такую задачу можно решить, используя библиотеку *React Animations* в качестве оболочки React для библиотеки Animate.css. Эта библиотека позволит эффективно добавлять анимационный стиль в разрабатываемые компоненты, не требуя создания дополнительных отрисовщиков и не увеличивая значительно размер генерируемой DOM-модели.

Эффективность этой библиотеки обеспечивается ее совмещением с какой-либо библиотекой типа CSS-in-JS. Технология библиотек CSS-in-JS позволяет размещать информацию о стилях непосредственно в коде JavaScript. Средство React также позволяет добавлять атрибуты стиля в виде компонентов React, но технология CSS-in-JS делает это более эффективно, динамически создавая общие элементы стиля в элементе `head` страницы.

На рынке доступно несколько библиотек CSS-in-JS, но для этого рецепта мы выберем библиотеку Radium (**https://oreil.ly/oNBEl**).

Для начала установим эту библиотеку, а также библиотеку React Animations:

```
$ npm install radium
$ npm install react-animations
```

Наше приложение будет исполнять анимационный эффект при каждом добавлении изображения в коллекцию (рис. 4.14).

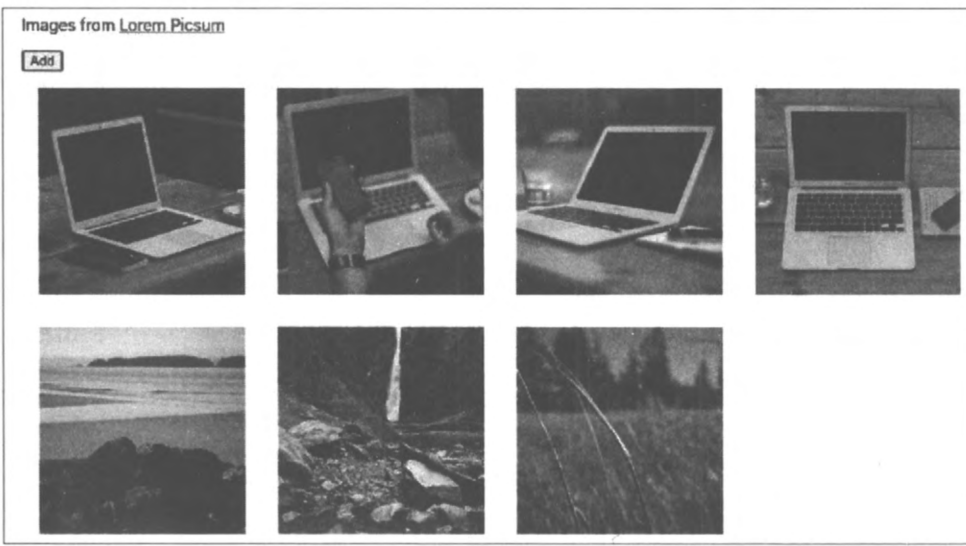


Рис. 4.14. Нажатие кнопки **Add** загружает новое изображение из хранилища pisum.photos

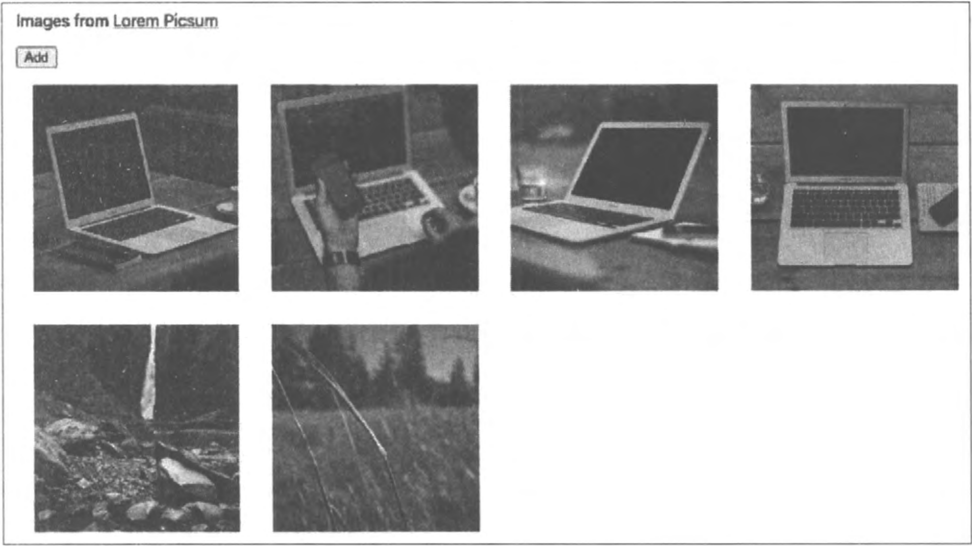


Рис. 4.15. Щелчок мышью по пятому изображению запускает анимационный эффект нарастающей прозрачности до полного исчезновения изображения и его удаления из коллекции

Аналогично при щелчке мышью по изображению оно удаляется из коллекции с эффектом нарастающей прозрачности (рис. 4.15)[4].

Начнем с импортирования некоторых анимаций из библиотеки React Animation и вспомогательного кода из библиотеки Radium (листинг 4.22).

Листинг 4.22. Импортирование кода из библиотек React Animation и Radium

```
import { pulse, zoomOut, shake, merge } from 'react-animations'
import Radium, { StyleRoot } from 'radium'
const styles = {
   created: {
      animation: 'x 0.5s',
      animationName: Radium.keyframes(pulse, 'pulse'),
  },
   deleted: {
      animation: 'x 0.5s',
      animationName: Radium.keyframes(merge(zoomOut, shake), 'zoomOut'),
   },
}
```

Из библиотеки React Animation импортируются анимационные эффекты `pulse`, `zoomOut` и `shake`. Анимация `pulse` будет реализована при добавлении изображений в коллекцию. А при удалении изображений будет использоваться объединение

[4] Бумажные книги, несомненно, прекрасные вещи, но чтобы полностью оценить анимационный эффект, загрузите код приложения с GitHub (**https://oreil.ly/OcAqo**).

анимаций zoomOut и shake. Для объединения анимаций служит функция merge библиотеки React Animation.

Функция styles генерирует все стили CSS, необходимые для исполнения всех этих полусекундных анимационных эффектов. Вся работа по исполнению анимаций выполняется функцией Radium.keyframes().

Нам нужно знать, когда действие анимации полностью завершится, потому что если мы удалим изображение перед этим, то у нас не будет, к чему применять анимационный эффект.

Для отслеживания CSS-анимаций можно передавать функцию обратного вызова onAnimationEnd любому элементу, который мы намереваемся анимировать. Для каждого элемента в нашей коллекции изображений нужно отслеживать три элемента:

- адрес URL представляемого им изображения;
- булеву переменную, которая будет иметь значение true, пока исполняется анимация создания изображения;
- булеву переменную, которая будет иметь значение true, пока исполняется анимация удаления изображения.

В листинге 4.23 приведен код для анимации изображений, добавляемых в коллекцию и удаляемых из нее.

Листинг 4.23. Код для анимации добавляемых и удаляемых изображений

```
import { useState } from 'react'
import { pulse, zoomOut, shake, merge } from 'react-animations'
import Radium, { StyleRoot } from 'radium'
import './App.css'
const styles = {
   created: {
      animation: 'x 0.5s',
      animationName: Radium.keyframes(pulse, 'pulse'),
   },
   deleted: {
      animation: 'x 0.5s',
      animationName: Radium.keyframes(merge(zoomOut, shake), 'zoomOut'),
   },
}

function getStyleForItem(item) {
   return item.deleting
      ? styles.deleted
      : item.creating
      ? styles.created
      : null
}
```

```
function App() {
  const [data, setData] = useState([])

  let deleteItem = (i) =>
    setData((d) => {
      const result = [...d]
      result[i].deleting = true
      return result
   })
  let createItem = () => {
    setData((d) => [
      ...d,
      {
        url: `https://picsum.photos/id/${d.length * 3}/200`,
        creating: true,
      },
    ])
  }
  let completeAnimation = (d, i) => {
    if (d.deleting) {
      setData((d) => {
        const result = [...d]
        result.splice(i, 1)
        return result
      })
    } else if (d.creating) {
      setData((d) => {
        const result = [...d]
        result[i].creating = false
        return result
      })
    }
 }
  return (
    <div className="App">
      <StyleRoot>
        <p>
          Images from 
          <a href="https://picsum.photos/">Lorem Picsum</a>
        </p>
        <button onClick={createItem}>Add</button>
        {data.map((d, i) => (
          <div
            style={getStyleForItem(d)}
            onAnimationEnd={() => completeAnimation(d, i)}
          >
```

```
                <img
                  id={`image${i}`}
                  src={d.url}
                  width={200}
                  height={200}
                  alt="Random"
                  title="Click to delete"
                  onClick={() => deleteItem(i)}
                />
              </div>
            ))}
          </StyleRoot>
        </div>
    )
}
export default App
```

Обсуждение

При выборе анимационного эффекта сначала нужно дать ответ на вопрос: что будет означать данная анимация?

Все анимационные эффекты должны иметь значение. Например, обозначать некое действие, связанное с бытием (создание или удаление), указывать изменение состояния (включение или отключение) или увеличивать либо уменьшать изображение, чтобы показать подробности или более общую картину соответственно. Или же иллюстрировать предел или границу (эффект отскакивания в конце длинного списка), или дать возможность пользователю выразить свое предпочтение (смахивание по сенсорному экрану влево или вправо).

Анимация также должна быть краткой. Большинство анимационных эффектов должны завершиться через полсекунды, чтобы пользователь мог ощутить сущность анимации без осознанного ее восприятия.

Анимационный эффект никогда не должен быть просто декоративным.

Исходный код данного рецепта можно загрузить на веб-сайте GitHub по адресу **https://oreil.ly/rRK8F**.

4.8. Анимация информационной графики посредством библиотеки TweenOne

ЗАДАЧА

Анимационные эффекты выполняются средствами CSS плавно и высокоэффективно. На стадии компоновки браузеры могут передавать CSS-анимацию графическому оборудованию, в результате чего анимационные эффекты будут не только

исполняться на уровне машинного кода, но и сам машинный код не создаст дополнительной нагрузки на центральный процессор.

Но в результате исполнения CSS-анимаций на графическом оборудовании код приложения "не знает", что происходит во время анимации. Он может отслеживать, когда анимация начинается, завершается или повторяется (события `onAnimationStart`, `onAnimationEnd` и `onAnimationIteration` соответственно), но все происходящее между началом и завершением будет загадкой.

При анимации, например, столбикового графика вдобавок к изменению высоты самих столбцов может быть желательным также анимировать числа соответствующих столбцов. Или в анимированном приложении для отслеживания прохождения велосипедной гонки одновременно с динамическим отображением положения отдельных велосипедистов на местности может требоваться отображать их текущую высоту над уровнем моря.

Но как создать анимации, состояние которых можно отслеживать в процессе исполнения?

РЕШЕНИЕ

Библиотека TweenOne создает анимационные эффекты при помощи JavaScript, что означает, что их можно отслеживать кадр за кадром.

Начнем с установки этой библиотеки:

```
$ npm install rc-tween-one
```

Библиотека TweenOne работает со стилями CSS, но не использует анимации CSS. Вместо этого она генерирует преобразования CSS, которые обновляет множество раз в секунду.

Элемент, который требуется анимировать, нужно заключить в оболочку элемента `<TweenOne/>`. Например, предположим, что нам нужно анимировать прямоугольник `rect` внутри графики SVG. Соответствующий код будет выглядеть следующим образом:

```
<TweenOne component='g' animation={...details here}>
   <rect width="2" height="6" x="3" y="-3" fill="white"/>
</TweenOne>
```

Элемент `TweenOne` принимает в качестве параметров имя анимируемого элемента и объект, описывающий требуемую анимацию. Мы рассмотрим этот объект более подробно чуть далее.

Имя элемента (в данном случае это просто буква `g`) используется библиотекой TweenOne для создания оболочки вокруг анимируемого объекта. Эта оболочка снабжается атрибутом стиля, содержащим набор преобразований CSS для перемещения и вращения содержимого.

Таким образом, на определенном этапе анимации в нашем примере DOM-модель может выглядеть следующим образом:

```
<g style="transform: translate(881.555px, 489.614px) rotate(136.174deg);">
   <rect width="2" height="6" x="3" y="-3" fill="white"/>
</g>
```

Хотя посредством библиотеки TweenOne можно создавать эффекты, похожие на анимации CSS, эта библиотека работает по-иному. Вместо того чтобы передавать исполнение анимации аппаратному обеспечению, она создает каждый кадр посредством JavaScript. Это имеет два последствия. Во-первых, она задействует вычислительные мощности центрального процессора (плохо), а во-вторых, позволяет отслеживать анимацию в процессе исполнения (хорошо).

Если `TweenOne` передать функцию обратного вызова `onUpdate`, на каждом кадре нам будет передаваться информация о процессе анимации:

```
<TweenOne component='g' animation={...details here} onUpdate={info=>{...}}>
   <rect width="2" height="6" x="3" y="-3" fill="white"/>
</TweenOne>
```

Переменная `ratio` передаваемого функции `onUpdate` объекта `info` может принимать значения от 0 до 1, представляющие пропорцию исполненной части анимации элемента `TweenOne` от общей ее длительности. Эту пропорцию можно использовать для анимации текста, связанного с графикой.

Например, если мы создадим анимированную панель управления, на которой отображается местонахождение автомобилей на гоночной трассе, то сможем посредством функции `onUpdate` отображать скорость и пройденное расстояние каждого автомобиля в процессе перемещения по трассе самого автомобиля.

Для этого примера будем применять графику SVG. Первым делом создадим строку, содержащую путь SVG, который представляет гоночную трассу:

```
export default 'm 723.72379,404.71306 ... -8.30851,-3.00521 z'
```

В действительности это значительно урезанная версия настоящей трассы, которую мы будем использовать. Строку трассы можно импортировать из файла track.js следующим образом:

```
import path from './track'
```

Отобразить трассу в компоненте React можно, выполнив отрисовку элемента `svg` (листинг 4.24).

Листинг 4.24. Отрисовка компонента svg

```
<svg height="600" width="1000" viewBox="0 0 1000 600"
     style={{backgroundColor: 'black'}}>
   <path stroke='#444' strokeWidth={10}
         fill='none' d={path}/>
</svg>
```

К этому коду можно добавить пару прямоугольников для представления автомобиля: красный — для кузова и белый — для лобового стекла (листинг 4.25).

Листинг 4.25. Добавление кода для представления автомобиля

```
<svg height="600" width="1000" viewBox="0 0 1000 600"
      style={{backgroundColor: 'black'}}>
   <path stroke='#444' strokeWidth={10}
         fill='none' d={path}/>
   <rect width={24} height={16} x={-12} y={-8} fill='red'/>
   <rect width={2} height={6} x={3} y={-3} fill='white'/>
</svg>
```

На рис. 4.16 показано, как будет выглядеть наша гоночная трасса с автомобилем в левом верхнем углу окна.

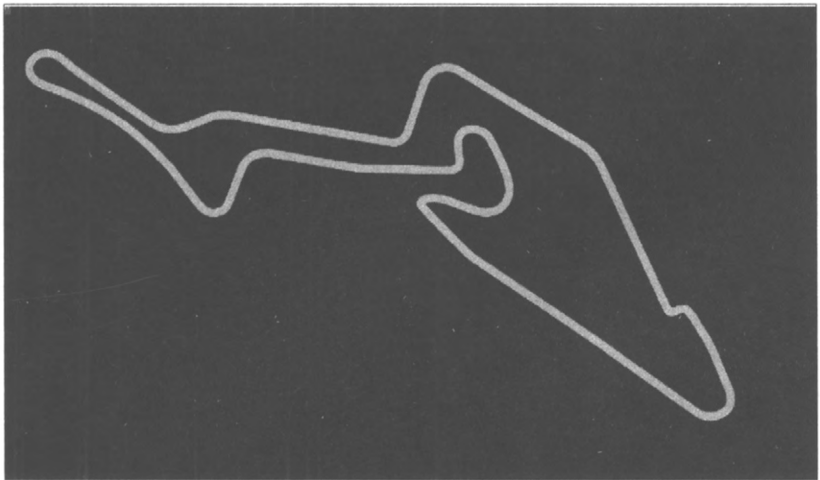


Рис. 4.16. Гоночная трасса с автомобилем в левом верхнем углу

Но как нам реализовать анимацию движения автомобиля по трассе? Это легко сделать при помощи подключаемого модуля PathPlugin библиотеки TweenOne, предназначенного для создания анимаций, следующих вдоль строк пути SVG:

```
import PathPlugin from 'rc-tween-one/lib/plugin/PathPlugin'
TweenOne.plugins.push(PathPlugin)
```

В результате мы сконфигурировали TweenOne для работы с анимациями путей SVG. Теперь мы можем рассмотреть, как описать анимацию для TweenOne. Делаем мы это посредством простого объекта JavaScript (листинг 4.26).

Листинг 4.26. Объект с описанием анимации

```
import path from './track'
const followAnimation = {
   path: { x: path, y: path, rotate: path },
   repeat: -1,
}
```

В этом объекте мы даем библиотеке TweenOne два указания:

1. Создавать перемещения и вращения, следующие вдоль строки пути, которую мы ранее импортировали из файла track.js.
2. Задав счетчику `repeat` значение `-1`, мы сообщаем, что анимация должна выполняться в бесконечном цикле.

Мы можем использовать этот объект в качестве основы для анимации нашего автомобиля (листинг 4.27).

Листинг 4.27. Базовый код для анимации автомобиля

```
<svg height="600" width="1000" viewBox="0 0 1000 600"
      style={{backgroundColor: 'black'}}>
   <path stroke='#444' strokeWidth={10}
         fill='none' d={path}/>
   <TweenOne component='g' animation={{...followAnimation, duration: 16000}}>
      <rect width={24} height={16} x={-12} y={-8} fill='red'/>
      <rect width={2} height={6} x={3} y={-3} fill='white'/>
   </TweenOne>
</svg>
```

Обратите внимание на наличие оператора расширения (троеточие) для предоставления дополнительного параметра `duration`. Его величина задана в миллисекундах, означая, что анимация должна длиться 16 секунд.

Мы можем добавить второй автомобиль и назначить функцию обратного вызова `onUpdate`, чтобы создать для каждого из них очень базовый набор имитированных телеметрических данных, когда они перемещаются по трассе. В листинге 4.28 приведен полный код приложения.

Листинг 4.28. Полный код анимации гонок и отображение телеметрических данных

```
import { useState } from 'react'
import TweenOne from 'rc-tween-one'
import Details from './Details'
import path from './track'
import PathPlugin from 'rc-tween-one/lib/plugin/PathPlugin'
import grid from './grid.svg'
import './App.css'
TweenOne.plugins.push(PathPlugin)
const followAnimation = {
   path: { x: path, y: path, rotate: path },
   repeat: -1,
}
function App() {
   const [redTelemetry, setRedTelemetry] = useState({
      dist: 0,
```

```
    speed: 0,
    lap: 0,
})
const [blueTelemetry, setBlueTelemetry] = useState({
    dist: 0,
    speed: 0,
    lap: 0,
})
const trackVehicle = (info, telemetry) => ({
    dist: info.ratio,
    speed: info.ratio - telemetry.dist,
    lap:
    info.ratio < telemetry.dist ? telemetry.lap + 1 : telemetry.lap,
})
return (
    <div className="App">
        <h1>Nürburgring</h1>
        <Details
            redTelemetry={redTelemetry}
        blueTelemetry={blueTelemetry}
    />
    <svg
        height="600"
        width="1000"
        viewBox="0 0 1000 600"
        style={{ backgroundColor: 'black' }}
    >
        <image href={grid} width={1000} height={600} />
        <path stroke="#444" strokeWidth={10} fill="none" d={path} />
        <path
            stroke="#c0c0c0"
            strokeWidth={2}
            strokeDasharray="3 4"
            fill="none"
            d={path}
        />
        <TweenOne
            component="g"
            animation={{
            ...followAnimation,
            duration: 16000,
            onUpdate: (info) =>
                setRedTelemetry((telemetry) =>
                    trackVehicle(info, telemetry)
                ),
            }}
        >
```

```
        <rect width={24} height={16} x={-12} y={-8} fill="red" />
        <rect width={2} height={6} x={3} y={-3} fill="white" />
      </TweenOne>

      <TweenOne
        component="g"
        animation={{
        ...followAnimation,
        delay: 3000,
        duration: 15500,
        onUpdate: (info) =>
          setBlueTelemetry((telemetry) =>
            trackVehicle(info, telemetry)
          ),
        }}
          >
            <rect width={24} height={16} x={-12} y={-8} fill="blue" />
            <rect width={2} height={6} x={3} y={-3} fill="white" />
          </TweenOne>
        </svg>
      </div>
  )
}
export default App
```

На рис. 4.17 показано, как выглядит исполняющееся приложение. Автомобили перемещаются вдоль пути гоночной трассы, поворачиваясь, чтобы передняя часть автомобиля была ориентирована по направлению движения.

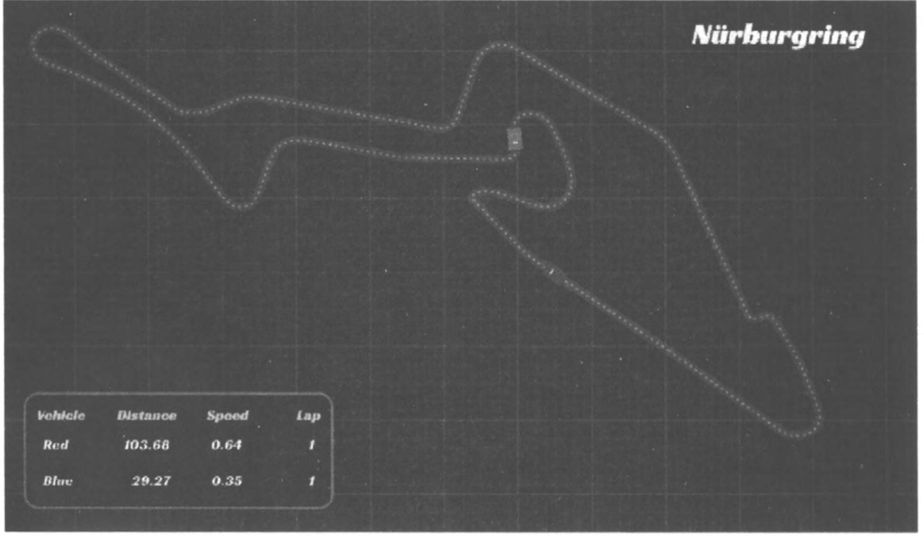


Рис. 4.17. Внешний вид конечной версии анимации с телеметрическими данными, генерируемыми из текущего состояния анимации

Обсуждение

Для реализации большинства анимационных эффектов пользовательского интерфейса следует использовать средства анимации CSS. Но в случае информационной графики часто требуется синхронизировать текст с графикой. Это можно реализовать при помощи средств библиотеки TweenOne, хотя за счет повышенной загрузки вычислительных мощностей центрального процессора.

Исходный код данного рецепта можно загрузить на веб-сайте GitHub по адресу **https://oreil.ly/8l7Vp**.

Подключение к службам

В отличие от фреймворков, например таких, как Angular, библиотека React не содержит всех возможностей, которые могут потребоваться для приложения. В частности, она не предоставляет стандартного способа получения приложениями данных от сетевых служб. Такая свобода действий является, в принципе, положительным аспектом, поскольку это означает, что приложения React могут использовать самые последние технологии. Но есть и недостаток — начинающим разработчикам React придется решать задачу выбора подходящего средства своими собственными силами.

В данной главе мы рассмотрим несколько способов подключения сетевых служб к разрабатываемым приложениям. Во всех этих рецептах мы увидим несколько общих идей и будем пытаться отделять код для работы с сетевыми возможностями от кода, использующего его компоненты. Таким образом, при появлении новой технологии сетевой службы мы сможем переключиться на нее без необходимости переделывать большой объем кода.

5.1. Преобразование сетевых вызовов в хуки

ЗАДАЧА

Одно из преимуществ разработки на основе компонентов заключается в том, что таким образом код разбивается на небольшие легко управляемые фрагменты, каждый из которых выполняет четко определенное, распознаваемое действие. В некотором смысле наилучший компонент — это компонент, который можно просматривать на большом экране без необходимости выполнять прокрутку. Одно из наилучших достоинств React состоит в том, что с течением времени эта библиотека стала проще во многих отношениях. Хуки React и отказ от компонентов на основе классов позволили избавиться от шаблонов и уменьшили объем кода.

Но добавление в компонент кода для сетевого обмена — один из верных способов увеличить размер компонента. Если вы стремитесь создавать простой код, следует стараться не включать сетевой код в разрабатываемые компоненты. Таким образом компоненты будут небольшого размера, а сетевой код будет больше поддаваться повторному использованию.

Так как же можно отделить сетевой код от остального кода компонента?

РЕШЕНИЕ

В этом рецепте мы рассмотрим способ, как переместить сетевые запросы в хуки React для отслеживания, продолжает ли исполняться сетевой запрос или же произошла какая-либо ошибка, не допустившая его успешного выполнения.

Прежде чем вникать в детали, рассмотрим, следующие три важных аспекта, которые нужно отслеживать при реализации асинхронного сетевого запроса:

- данные, возвращаемые запросом;
- ход загрузки с сервера запрошенных данных;
- любые ошибки, которые могли произойти при исполнении запроса.

Мы увидим все три аспекта присутствующими в каждом из рецептов этой главы. Не важно, исполняем ли мы запрос посредством команды `fetch` или `axios`, через связующее программное обеспечение Redux или при помощи промежуточного языка запросов для API, например GraphQL, для запроса всегда будут иметь важность данные, состояние их загрузки и ошибки процесса загрузки.

Для примера создадим простую доску сообщений, состоящую из нескольких форумов. Сообщения форумов состоят из поля автора сообщения и поля текста сообщения. На рис. 5.1 приведен снимок экрана этого приложения, которое можно загрузить на веб-сайте GitHub.

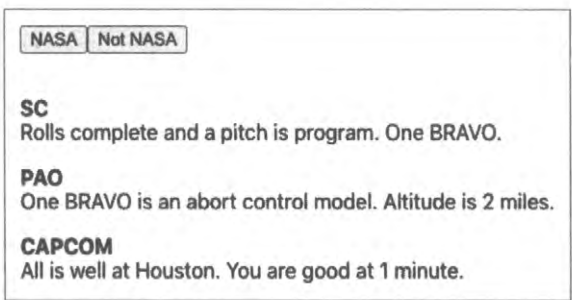


Рис. 5.1. Кнопки позволяют выбрать форум NASA или форумы, иные, чем NASA

Кнопки в верхней части окна приложения позволяют выбрать форум NASA (NASA) или форумы, иные, чем NASA (Not NASA). Бэкенд обработка для нашего приложения обеспечивается сервером Node, а в само приложение было загружено несколько сообщений для форума NASA. Загрузите исходный код приложения и активизируйте сервер бэкенда, запустив на исполнение файл сценария server.js в корневой папке приложения:

```
$ node ./server.js
```

Сервер бэкенда будет исполняться по адресу **http://localhost:5000**. Теперь можно запустить на исполнение само приложение React:

```
$ npm run start
```

Приложение, как обычно, исполняется на порту 3000.

В режиме разработки все запросы к серверу бэкенда проходят через прокси-сервер React. При создании приложения посредством команды `create-react-app` эту возможность можно реализовать, добавив свойство `proxy` в файле package.json и присвоив ему значение `http://localhost:5000`. В результате сервер React будет передавать запросы API нашему серверу бэкенда server.js. Например, запрос `http://localhost:3000/messages/nasa` (который возвращает массив сообщений из форума NASA) будет перенаправлен на адрес прокси-сервера http://localhost:5000/messages/nasa.

Сетевой запрос для чтения сообщений подается посредством простой команды `fetch` (листинг 5.1).

Листинг 5.1. Сетевой запрос для чтения сообщений

```
const response = await fetch(`/messages/${forum}`)
if (!response.ok) {
   const text = await response.text()
   throw new Error(`Unable to read messages for ${forum}: ${text}`)
}
const body = await response.json()
```

Переменной `forum` в этом коде будет присвоено строковое значение идентификатора форума. Команда `fetch` — асинхронного типа и возвращает обещание (promise), поэтому будем ожидать его получения командой `await`. Затем мы можем проверить, не завершился ли вызов сбоем с каким-либо кодом ошибки HTTP, и если да, то генерируем исключение. Из ответа мы извлечем объект JSON и сохраним его в переменной `body`. В случае отсутствия в теле ответа объекта JSON правильного формата также генерируется исключение.

Нам нужно отслеживать три вещи: данные, состояние загрузки и наличие любых ошибок. Обработку всего этого мы заключим в специальный хук, поэтому создадим три состояния: `data`, `loading` и `error` (листинг 5.2).

Листинг 5.2. Код хука useMessages

```
const useMessages = (forum) => {
   const [data, setData] = useState([])
   const [loading, setLoading] = useState(false)
   const [error, setError] = useState()
   ....
   return { data, loading, error }
}
```

В хук `useMessages` в качестве параметра будем передавать название форума, а он будет возвращать объект, содержащий состояния данных `data`, загрузки `loading` и ошибки `error`. Значения из объекта можно извлечь и переименовать в любом компоненте посредством деструктурирования объекта, как показано в листинге 5.3.

Листинг 5.3. Деструктурирование объекта

```
const {
  data: messages,
  loading: messagesLoading,
  error: messagesError,
} = useMessages('nasa')
```

Переименование переменных в операторе расширения помогает избежать конфликтов имен. Например, если нужно прочитать сообщения из нескольких форумов, можно выполнить другой вызов хука `useMessages` и переменную для его ответа назвать не `messages`, а по-другому.

Но возвратимся обратно к нашему хуку `useMessages`. Сетевой запрос зависит от значения передаваемой ему переменной `forum`, поэтому необходимо обеспечить исполнение запроса `fetch` внутри хука `useEffect` (листинг 5.4).

Листинг 5.4. Помещение запроса в оболочку хука `useEffect`

```
useEffect(() => {
  setError(null)
  if (forum) {
    ....
  } else {
    setData([])
    setLoading(false)
  }
}, [forum])
```

На данном этапе мы опускаем код, который выполняет сам запрос. Код в хуке `useEffect` будет исполняться при первом вызове этого хука. При повторной отрисовке клиентского компонента и передаче такого же самого значения `forum` хук `useEffect` исполняться не будет, поскольку зависимость `[forum]` не изменится. Хук будет исполняться повторно только при изменении значения переменной `forum`.

Теперь рассмотрим, как можно вставить запрос `fetch` к этому хуку (листинг 5.5).

Листинг 5.5. Добавление запроса `fetch` к хуку `useEffect`

```
import { useEffect, useState } from 'react'
const useMessages = (forum) => {
  const [data, setData] = useState([])
  const [loading, setLoading] = useState(false)
  const [error, setError] = useState()
  useEffect(() => {
    let didCancel = false
    setError(null)
```

```
      if (forum) {
         ;(async () => {
            try {
               setLoading(true)
               const response = await fetch(`/messages/${forum}`)
               if (!response.ok) {
                  const text = await response.text()
                  throw new Error(
                     `Unable to read messages for ${forum}: ${text}`
                  )
               }
               const body = await response.json()
               if (!didCancel) {
                  setData(body)
               }
            } catch (err) {
               setError(err)
            } finally {
               setLoading(false)
            }
         })()
      } else {
         setData([])
         setLoading(false)
      }
      return () => {
         didCancel = true
      }
   }, [forum])
   return { data, loading, error }
}
export default useMessages
```

Поскольку для правильной обработки обещаний мы используем функцию `wait`, нам нужно обернуть код в довольно неприглядный вызов `(async () => {...})`. Внутри этой оболочки мы сможет задать значения для переменных `data`, `loading` и `error`, пока запрос исполняется, завершается и (возможно) возвращает недействительные данные. Все это происходит асинхронно после завершения вызова хука. При изменении состояния переменных `data`, `loading` и `error` хук вызовет повторную отрисовку компонента с этими новыми значениями.

Поскольку хук содержит асинхронный код, возврат хука происходит до получения ответа на сетевой запрос. Это означает возможный повторный вызов хука до получения ответа на предыдущий сетевой запрос. Во избежание разрешения сетевых ответов в неправильном порядке код примера отслеживает, не был ли текущий запрос замещен более поздним запросом, используя для этого переменную `didCancel`. На основе этой переменной будет приниматься решение, возвращать ли данные из хука. При этом сам сетевой запрос не будет отменяться. Вопрос отмены запроса рассматривается в *разделе 5.3*.

Теперь рассмотрим содержимое файла App.js примера приложения, чтобы разобраться с использованием этого хука (листинг 5.6).

Листинг 5.6. Содержимое файла App.js примера приложения

```
import './App.css'
import { useState } from 'react'
import useMessages from './useMessages'
function App() {
   const [forum, setForum] = useState('nasa')
   const {
      data: messages,
      loading: messagesLoading,
      error: messagesError,
   } = useMessages(forum)
   return (
      <div className="App">
         <button onClick={() => setForum('nasa')}>NASA</button>
         <button onClick={() => setForum('notNasa')}>Not NASA</button>
         {messagesError ? (
            <div className="error">
               Something went wrong:
               <div className="error-contents">
                  {messagesError.message}
               </div>
            </div>
         ) : messagesLoading ? (
            <div className="loading">Loading...</div>
         ) : messages && messages.length ? (
            <dl>
               {messages.map((m) => (
                  <>
                     <dt>{m.author}</dt>
                     <dd>{m.text}</dd>
                  </>
               ))}
            </dl>
         ) : (
            'No messages'
         )}
      </div>
   )
}
export default App
```

Наше приложение при нажатии кнопки **NASA** или **Not NASA** отображает в своем окне соответствующий форум. При нажатии кнопки **Not NASA** сервер возвращает

ошибку 404, в результате чего в окне приложения отображается сообщение об ошибке. На рис. 5.2 показано, как наше приложение отображает этап загрузки сообщений для форума *NASA* (вверху), загруженные сообщения (посередине) и сообщение об ошибке при попытке загрузить данные из отсутствующего форума *Not NASA* (внизу).

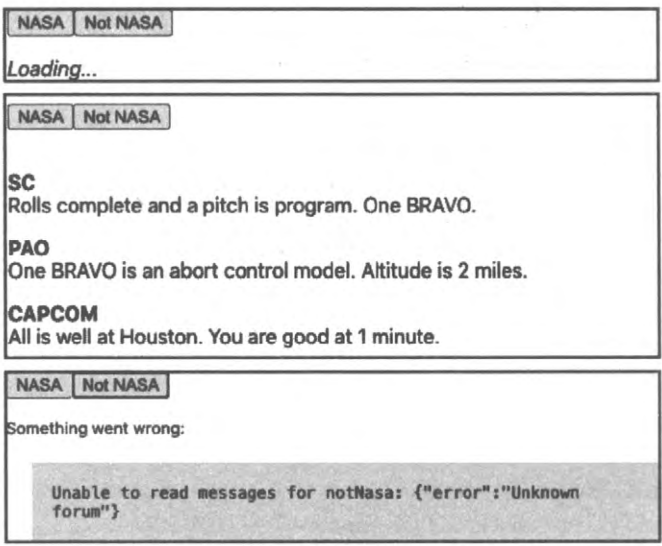


Рис. 5.2. Отображение процесса загрузки сообщений, загруженных сообщений и сообщения об ошибке

Хук `useMessages` также обрабатывает и возвращаемые сервером ошибки, как показано на рис. 5.3.

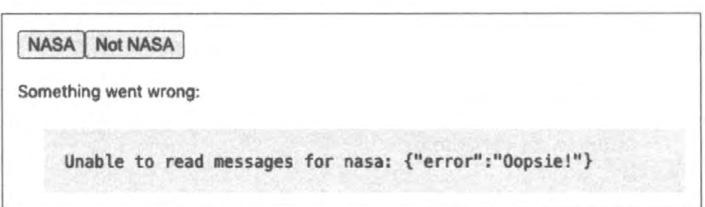


Рис. 5.3. Компонент может отображать возвращаемые сервером сообщения об ошибках

Обсуждение

При создании приложения может возникнуть соблазн разрабатывать код без дополнительных функций, неявно предполагая, что все будет работать должным образом без нештатных ситуаций. Но все же полезно уделить некоторое время разработке кода для обработки ошибок, а также для отображения процесса загрузки данных. Это сделает ваше приложение более удобным в использовании и облегчит обнаружение медленных служб и ошибок.

Можно также рассмотреть возможность совмещения этого рецепта с рецептом *раздела 4.1*, чтобы пользователям было легче описать, что произошло.

Исходный код данного рецепта можно загрузить на веб-сайте GitHub по адресу **https://oreil.ly/T6M6q**.

5.2. Автоматическое обновление посредством счетчиков состояния

ЗАДАЧА

Сетевым службам часто требуется взаимодействовать друг с другом. Возьмем, например, приложение форума, рассмотренное в *разделе 5.1*. Мы хотим, чтобы список сообщений обновлялся автоматически при каждом добавлении в него нового сообщения.

Для этого приложения мы уже создали в том разделе хук `useMessages`, содержащий весь требуемый код для чтения сообщений форума.

Теперь мы добавим в это приложение форму для отправки сообщений на сервер (листинг 5.7).

Листинг 5.7. Форма для отправки сообщений на сервер

```
const {
   data: messages,
   loading: messagesLoading,
   error: messagesError,
} = useMessages('nasa')
const [text, setText] = useState()
const [author, setAuthor] = useState()
const [createMessageError, setCreateMessageError] = useState()
// Other code here...
<input
   type="text"
   value={author}
   placeholder="Author"
   onChange={(evt) => setAuthor(evt.target.value)}
/>
<textarea
   value={text}
   placeholder="Message"
   onChange={(evt) => setText(evt.target.value)}
/>
<button
   onClick={async () => {
      try {
         await [code to post message here]
```

```
         setText('')
         setAuthor('')
      } catch (err) {
         setCreateMessageError(err)
      }
   }}
>
   Post
</button>
```

Но эта форма имеет один недостаток: новые сообщения не отображаются в списке, пока страница не будет обновлена вручную (рис. 5.4).

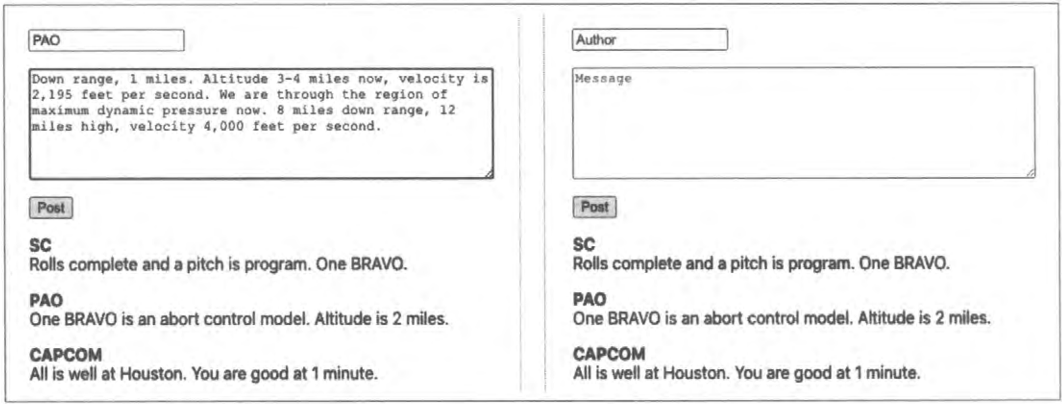


Рис. 5.4. Отправленные сообщения не отображаются в списке сообщений

Итак, как нам автоматически загружать сообщения с сервера после сохранения на нем нового сообщения?

РЕШЕНИЕ

Активирование операции обновления данных будем выполнять при помощи счетчика состояний. Счетчик состояний — это просто постоянно нарастающее число. Текущее значение счетчика не важно, нужно лишь изменять его каждый раз, когда требуется перезагрузить данные:

```
const [stateVersion, setStateVersion] = useState(0)
```

Счетчик состояний можно рассматривать как представление воспринимаемой нами версии данных на сервере. Когда мы делаем что-либо, что может изменить состояние сервера, мы обновляем счетчик состояний, чтобы отображать это изменение:

```
// code to post a new message here
   (здесь размещается код для отправки нового сообщения)
setStateVersion((v) => v + 1)
```

Обратите внимание на то, что мы инкрементируем значение переменной `stateVersion` при помощи функции, а не простым оператором `setStateVersion(stateVersion + 1)`. Если новое значение зависит от старого, то значение состояния всегда нужно обновлять при помощи функции, поскольку состояния в React устанавливаются асинхронно. Поэтому если исполнить оператор `setStateVersion(stateVersion + 1)` дважды в быстрой последовательности, то значение `stateVersion` может не измениться между этими двумя операциями и мы можем пропустить его инкрементацию.

Код, считывающий текущий набор сообщений, заключен в хук `useEffect`, который мы можем заставить выполнять возврат, сделав его зависимым от значения переменной `stateValue` (листинг 5.8).

Листинг 5.8. Принуждаем возврат хука `useEffect`

```
useEffect(() => {
   setError(null)
   if (forum) {
      // Code to read /messages/:forum
   } else {
      setData([])
      setLoading(false)
   }
}, [forum, stateVersion])
```

Таким образом, если изменится значение переменной `forum` или переменной `stateVersion`, хук автоматически перезагрузит сообщения (рис. 5.5).

Вот таким будет наш подход. Теперь нам нужно определить, куда мы вставим этот код. В листинге 5.9 приведена предыдущая версия компонента, которая только считывает сообщения.

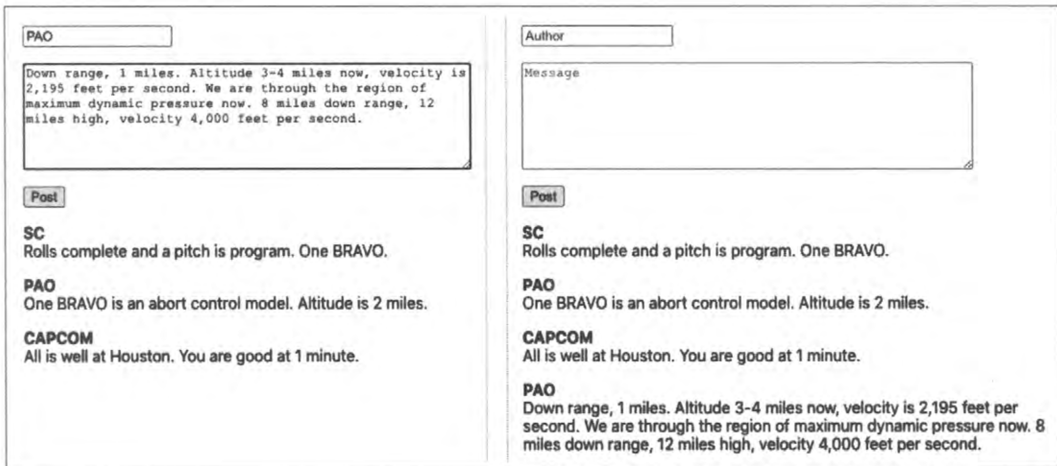


Рис. 5.5. Отправка нового сообщения вызывает обновление списка сообщений

Листинг 5.9. Версия компонента, только считывающая сообщения

```
import './App.css'
import { useState } from 'react'
import useMessages from './useMessages'
function App() {
  const [forum, setForum] = useState('nasa')
  const {
    data: messages,
    loading: messagesLoading,
    error: messagesError,
  } = useMessages(forum)
  return (
    <div className="App">
      <button onClick={() => setForum('nasa')}>NASA</button>
      <button onClick={() => setForum('notNasa')}>Not NASA</button>
      {messagesError ? (
        <div className="error">
          Something went wrong:
          <div className="error-contents">
            {messagesError.message}
          </div>
        </div>
      ) : messagesLoading ? (
        <div className="loading">Loading...</div>
      ) : messages && messages.length ? (
        <dl>
          {messages.map((m) => (
            <>
              <dt>{m.author}</dt>
              <dd>{m.text}</dd>
            </>
          ))}
        </dl>
      ) : (
        'No messages'
      )}
    </div>
  )
}
export default App
```

К этому компоненту мы добавим нашу новую форму. В этот же компонент можно было бы также включить код для работы в сети и код счетчика состояний. Но тогда код отправки сообщений оказался бы в этом компоненте, а код для их считывания — в хуке `useMessages`. Но будет лучше, чтобы весь код для работы в сети нахо-

дился в хуке. Таким образом, не только наш компонент будет аккуратнее оформлен, но и код для сетевых операций будет легче использовать повторно.

В листинге 5.10 приведен код новой версии хука `useMessages`, переименованного в `useForum`[1].

Листинг 5.10. Код хука `useForum`

```
import { useCallback, useEffect, useState } from 'react'
const useForum = (forum) => {
   const [data, setData] = useState([])
   const [loading, setLoading] = useState(false)
   const [error, setError] = useState()
   const [creating, setCreating] = useState(false)
   const [stateVersion, setStateVersion] = useState(0)
   const create = useCallback(
   async (message) => {
      try {
         setCreating(true)
         const response = await fetch(`/messages/${forum}`, {
            method: 'POST',
            body: JSON.stringify(message),
            headers: {
               'Content-type': 'application/json; charset=UTF-8',
            },
         })
         if (!response.ok) {
            const text = await response.text()
            throw new Error(
               `Unable to create a ${forum} message: ${text}`
            )
         }
         setStateVersion((v) => v + 1)
      } finally {
         setCreating(false)
      }
   },
   [forum]
)
useEffect(() => {
   let didCancel = false
   setError(null)
```

[1] Переименование хука объясняется тем, что он теперь используется не только для чтения списка сообщений, но для работы с форумом в общем. В дальнейшем к нему также можно добавить функции для удаления и редактирования сообщений, а также для пометки их флагом.

```
   if (forum) {
      ;(async () => {
         try {
            setLoading(true)
            const response = await fetch(`/messages/${forum}`)
            if (!response.ok) {
               const text = await response.text()
               throw new Error(
                  `Unable to read messages for ${forum}: ${text}`
               )
            }
            const body = await response.json()
            if (!didCancel) {
               setData(body)
            }
         } catch (err) {
            setError(err)
         } finally {
            setLoading(false)
         }
      })()
   } else {
      setData([])
      setLoading(false)
   }
   return () => {
      didCancel = true
   }
}, [forum, stateVersion])

return { data, loading, error, create, creating }
}
export default useForum
```

Теперь в хуке `useForum` мы создаем функцию `create`, после чего возвращаем его компоненту вместе с разными другими составляющими состояния. Обратите внимание на то, что эту функцию `create` мы заключаем в хук `useCallback`, что означает, что мы не будем создавать новую версию этой функции, если только это не потребуется для того, чтобы создать данные для другого значения `forum`.

Будьте осторожны при создании функций внутри хуков и компонентов. Средство React часто инициирует повторную отрисовку при создании нового объекта функции, даже если данная функции делает то же самое, что и предыдущая версия.

При вызове функции `create` она отправляет новое сообщение на форум, а затем обновляет значение переменной `stateVersion`, что автоматически принудит хук считать сообщения с сервера. Обратите также внимание на присутствие булевой пере-

менной `creating`, которая имеет значение `true`, пока сетевой код отправляет сообщение на сервер. Эта переменная служит для отключения кнопки **POST**.

Но мы не следим за тем, возникают ли какие-либо ошибки в функции `create`. Почему нет? В конце концов, мы делаем это при чтении данных с сервера. Потому что при изменении данных на сервере часто требуется более высокий уровень контроля над обработкой ошибок, чем при простом считывании данных. В примере приложения при отправке сообщения на сервер мы очищаем форму сообщений. В случае возникновения ошибки мы хотим, чтобы сообщение не удалялось из формы.

Теперь рассмотрим код, вызывающий наш хук (листинг 5.11).

Листинг 5.11. Код для вызова хука `useForum`

```
import './App.css'
import { useState } from 'react'
import useForum from './useForum'
function App() {
   const {
      data: messages,
      loading: messagesLoading,
      error: messagesError,
      create: createMessage,
      creating: creatingMessage,
   } = useForum('nasa')
   const [text, setText] = useState()
   const [author, setAuthor] = useState()
   const [createMessageError, setCreateMessageError] = useState()
   return (
      <div className="App">
         <input
            type="text"
            value={author}
            placeholder="Author"
            onChange={(evt) => setAuthor(evt.target.value)}
         />
         <textarea
            value={text}
            placeholder="Message"
            onChange={(evt) => setText(evt.target.value)}
         />
         <button
            onClick={async () => {
            try {
               await createMessage({ author, text })
               setText('')
               setAuthor('')
```

```
            } catch (err) {
               setCreateMessageError(err)
            }
         }}
         disabled={creatingMessage}
      >
         Post
      </button>
      {createMessageError ? (
         <div className="error">
            Unable to create message
            <div className="error-contents">
               {createMessageError.message}
            </div>
         </div>
      ) : null}
      {messagesError ? (
         <div className="error">
            Something went wrong:
            <div className="error-contents">
               {messagesError.message}
            </div>
         </div>
      ) : messagesLoading ? (
         <div className="loading">Loading...</div>
      ) : messages && messages.length ? (
         <dl>
            {messages.map((m) => (
               <>
                  <dt>{m.author}</dt>
                  <dd>{m.text}</dd>
               </>
            ))}
         </dl>
      ) : (
         'No messages'
      )}
      </div>
   )
}
export default App
```

Подробности считывания и записи сообщений скрыты внутри хука `useForum`. Для присвоения функции `create` переменной `createMessage` мы используем деструктурирование объекта. При вызове функции `createMessage` она не только отправит сооб-

щение на сервер, но также автоматически прочитает новые сообщения форума и обновит содержимое окна соответствующим образом (рис. 5.6).

Теперь наш хук не просто является средством для считывания данных с сервера, а становится службой для управления самим форумом.

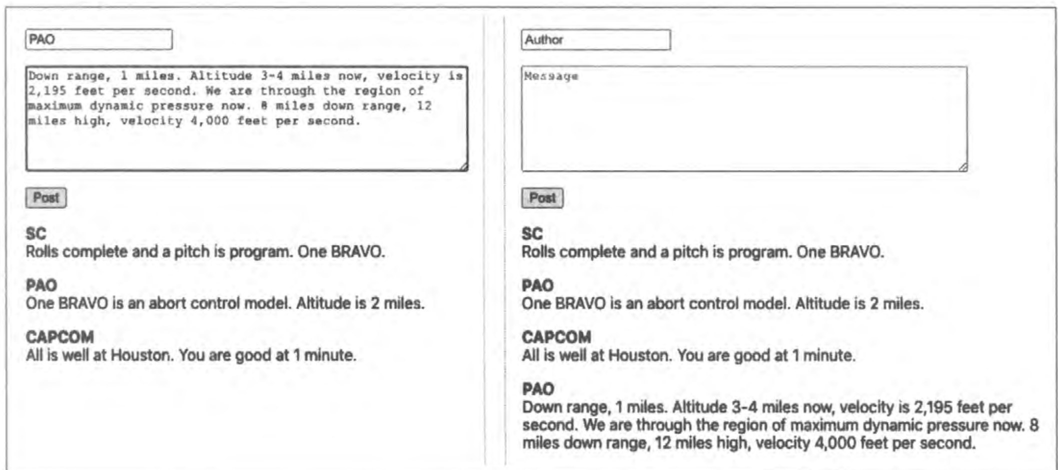


Рис. 5.6. Отправление нового сообщения на сервер и автоматическое обновление содержимого окна

Обсуждение

Будьте осторожны, если вы намереваетесь применять описанный подход для сохранения данных на сервере посредством *одного* компонента, а затем считывать их с сервера при помощи *другого*. Экземпляры отдельных хуков будут использовать отдельные счетчики состояний, и сохранение данных в одном компоненте не вызовет автоматического чтения данных в другом компоненте. Если вы хотите разделить код для сохранения и чтения данных между отдельными компонентами, вызывайте специальный хук в каком-либо общем родительском компоненте, передайте данные и отправьте функции дочерним компонентам, для которых они требуются.

Если требуется выполнять регулярные опросы сетевой службы, рассмотрите возможность создания часов и сделайте код для сетевых операций зависимым от текущего значения часов, подобно тому, как код в этом рецепте зависит от счетчика состояний[2].

Исходный код данного рецепта можно загрузить на веб-сайте GitHub по адресу **https://oreil.ly/knyC5**.

---

[2] См. *раздел 3.4*.

5.3. Отмена сетевых запросов посредством маркеров

ЗАДАЧА

Рассмотрим следующее неверно функционирующее приложение, выполняющее поиск городов. Когда пользователь начинает вводить название города в поле поиска, в окне приложения начинает отображаться список совпадающих городов. Например, при вводе букв "C... H... I. G..." в таблице результатов отображаются совпадающие города. Но затем, после кратковременной правильной работы, в таблице результатов отображается более длинный список городов, содержащий неправильные результаты, например, Wichita Falls (рис. 5.7).

Рис. 5.7. На первых порах поиск работает должным образом (слева), но затем выводится список, содержащий неправильные города

Проблема состоит в том, что система отправляет новый сетевой запрос при вводе пользователем каждой новой буквы. Но исполнение сетевых запросов занимает разное время. В данном примере выполнение сетевого запроса на поиск строки "CHI" заняло на пару секунд больше, чем на поиске строки "CHIG". Вследствие этого результаты поиска строки "CHI" были возвращены после результатов поиска строки "CHIG".

Как же тогда можно предотвратить возврат результатов асинхронных сетевых запросов в неправильном порядке?

РЕШЕНИЕ

При отправке сетевому серверу многократных запросов `GET` старые запросы можно отменить, прежде чем отправлять новые. Таким образом, мы никогда не получим результаты запросов в неправильном порядке, поскольку одновременно к службе будет обращаться только один запрос.

Для этого рецепта нам потребуется сетевая библиотека Axios. Установим ее как обычно:

```
$ npm install axios
```

Библиотека Axios служит оболочкой для нативной функции `fetch` и предоставляет возможность отменять сетевые запросы, используя маркеры. Реализация библиотеки Axios основана на проектной инициативе отменяемых обещаний, предложенной организацией ECMA (**https://oreil.ly/jd4LF**).

Начнем с исследования нашего проблемного кода (листинг 5.12). Код для сетевых операций заключен в специализированный хук[3].

Листинг 5.12. Код проблемного приложения

```
import { useEffect, useState } from 'react'
import axios from 'axios'
const useSearch = (terms) => {
  const [data, setData] = useState([])
  const [loading, setLoading] = useState(false)
  const [error, setError] = useState()
  useEffect(() => {
    setError(null)
    if (terms) {
      ;(async () => {
        try {
          setLoading(true)
          const response = await axios.get('/search', {
            params: { terms },
          })
          setData(response.data)
        } catch (err) {
          setError(err)
        } finally {
          setLoading(false)
        }
      })()
    } else {
      setData([])
      setLoading(false)
    }
  }, [terms])
  return { data, loading, error }
}
export default useSearch
```

Искомая строка содержится в параметре `terms`. Причина проблемы следующая. Сначала код подал сетевой запрос к /search для поиска строки "CHI".

---

[3] Сравните этот код с кодом в *разделе 5.1*, в котором используется команда `fetch`.

Пока этот запрос находился в процессе исполнения, был сделан другой запрос поиска, на этот раз строки "CHIG". Выполнение первого запроса заняло больше времени, чем второго, что и вызвало ошибку.

Для устранения этой проблемы мы воспользуемся маркером отмены библиотеки Axios. Если пометить запрос маркером, то в дальнейшем запрос можно отменить при помощи этого маркера. Браузер завершит исполнение запроса, и он пропадет без следа.

Чтобы использовать маркер, сначала нужно создать для него источник:

```
const source = axios.CancelToken.source()
```

Источник маркера можно сравнить с пультом дистанционного управления для сетевого запроса. После подключения сетевого запроса к источнику последнему можно дать указание отменить сетевой запрос. Источник ассоциируется с запросом посредством команды `source.token`, как показано в листинге 5.13.

Листинг 5.13. Ассоциирование источника маркера с запросом

```
const response = await axios.get('/search', {
  params: { terms },
  cancelToken: source.token,
})
```

Система Axios запомнит, какой маркер с каким сетевым запросом связан. Отмена сетевого запроса осуществляется следующим образом:

```
source.cancel('axios request canceled')
```

Но отменять запрос обязательно нужно только тогда, когда мы отправляем новый запрос. К счастью, наш сетевой вызов заключен в оболочку хука `useEffect`, который обладает одной полезной возможностью. В частности, при возвращении функции, которая отменяет текущий запрос, эта функция будет исполнена непосредственно перед следующим исполнением хука `useEffect`. Таким образом, при возврате функции, отменяющей текущий сетевой запрос, мы автоматически отменяем старый сетевой запрос перед каждым исполнением нового запроса[4]. В листинге 5.14 приведена обновленная версия специализированного хука.

Листинг 5.14. Обновленная версия специализированного хука

```
import { useEffect, useState } from 'react'
import axios from 'axios'
const useCancelableSearch = (terms) => {
  const [data, setData] = useState([])
  const [loading, setLoading] = useState(false)
  const [error, setError] = useState()
```

[4] Если предыдущий сетевой запрос завершен, его отмена не повлечет за собой никаких последствий.

```
  useEffect(() => {
    setError(null)
    if (terms) {
      const source = axios.CancelToken.source()
      ;(async () => {
        try {
          setLoading(true)
          const response = await axios.get('/search', {
            params: { terms },
            cancelToken: source.token,
          })
          setData(response.data)
        } catch (err) {
          setError(err)
        } finally {
          setLoading(false)
        }
      })()
      return () => {
        source.cancel('axios request cancelled')
      }
    } else {
      setData([])
      setLoading(false)
    }
  }, [terms])

  return { data, loading, error }
}
export default useCancelableSearch
```

Обсуждение

Этот подход к отмене сетевых запросов следует применять только при доступе к идемпотентным службам. На практике это означает, что его следует использовать для запросов GET, где вас интересуют только самые последние результаты.

Исходный код данного рецепта можно загрузить на веб-сайте GitHub по адресу **https://oreil.ly/aQj5g**.

5.4. Сетевые вызовы посредством Redux

ЗАДАЧА

Библиотека Redux предоставляет возможность централизованного управления состоянием приложения[5]. Изменения состояния приложения осуществляются от-

[5] При первом знакомстве эта библиотека может выглядеть довольно сложной. Дополнительные рецепты с использованием библиотеки Redux приведены в *главе 3*.

правкой команд (называющихся действиями — *actions*), которые захватываются и обрабатываются функциями JavaScript, называющимися *преобразователями* (или редюсерами — reducers). Библиотека Redux пользуется популярностью среди программистов React, поскольку она предоставляет возможность отделить код логики управления состояниям от кода компонента. Команды Redux исполняются асинхронно, но в строгом порядке, что позволяет создавать большие и сложные приложения с использованием Redux, которые одновременно и эффективные, и стабильные.

Было бы просто замечательно, если бы эти мощные возможности Redux можно было применить для управления всеми нашими сетевыми запросами. Мы могли бы отправлять действия, указывающие, например "Прочитать результаты последнего поиска", а Redux мог бы подать соответствующий сетевой запрос, а затем обновить должным образом общее состояние.

Но, чтобы обеспечить стабильность работы кода Redux, преобразующие функции должны удовлетворять нескольким довольно строгим критериям, одним из которых является отсутствие побочных эффектов. Это означает полный запрет на подачу сетевых запросов изнутри преобразователя.

Но если нам нельзя осуществлять сетевые запросы изнутри функций-преобразователей, то как можно сконфигурировать Redux для взаимодействия с сетью?

РЕШЕНИЕ

В приложениях React, использующих возможности библиотеки Redux, компоненты публикуют (отправляют) действия, а преобразователи реагируют на действия обновлением общего состояния (рис. 5.8).

Рис. 5.8. Обновление общего состояния посредством преобразователей Redux

Если требуется создавать действия с дополнительными эффектами, нужно использовать промежуточное программное обеспечение Redux. Промежуточное ПО принимает действия перед тем, как Redux отправляет их преобразователям, и может преобразовывать или удалять их либо создавать новые действия. Самое главное, коду промежуточного ПО Redux разрешается иметь побочные эффекты. Это означает, что если компонент отправляет действие, указывающее "Найти эту строку", мы можем создать промежуточное ПО, которое принимает это действие, генерирует сетевой вызов, а затем преобразует ответ в новое действие "Сохранить эти ре-

зультаты поиска". На рис. 5.9 приводится блок-схема, демонстрирующая принцип работы промежуточного программного обеспечения Redux.

Создадим теперь промежуточное ПО для перехвата действия типа SEARCH и используем его для создания сетевой службы.

Рис. 5.9. Реализация сетевых вызовов посредством промежуточного ПО Redux

Получив ответ на сетевой запрос с результатами поиска, создадим новое действие типа SEARCH_RESULTS, которое даст возможность сохранения результатов поиска в общем состоянии Redux. Объект нашего действия будет выглядеть, как показано в листинге 5.15.

Листинг 5.15. Код объекта действия SEARCH

```
{
  "type": "SEARCH",
  "payload": "Some search text"
}
```

А в листинге 5.16 приведен код из файла axiosMiddlewar.js, при помощи которого мы будем перехватывать действия SEARCH.

Листинг 5.16. Код для перехвата действий SEARCH

```
import axios from 'axios'
const axiosMiddleware = (store) => (next) => (action) => {
  if (action.type === 'SEARCH') {
    const terms = action.payload
    if (terms) {
      ;(async () => {
        try {
          store.dispatch({
            type: 'SEARCH_RESULTS',
            payload: {
              loading: true,
              data: null,
```

```
                error: null,
             },
          })
          const response = await axios.get('/search', {
             params: { terms },
          })
          store.dispatch({
             type: 'SEARCH_RESULTS',
             payload: {
                loading: false,
                error: null,
                data: response.data,
             },
          })
       } catch (err) {
          store.dispatch({
             type: 'SEARCH_RESULTS',
             payload: {
                loading: false,
                error: err,
                data: null,
             },
          })
       }
     })()
   }
 }
 return next(action)
}
export default axiosMiddleware
```

Выражение функции для промежуточного ПО Redux может быть не совсем понятным. Его можно рассматривать как функцию, которая в качестве аргументов принимает хранилище, действие и другую функцию, называющуюся `next`, которая может перенаправить действие остальным частям Redux.

В коде в листинге 5.16 выполняется проверка, не появилось ли действие типа `SEARCH`. Если да, то выполняется сетевой вызов. В противном случае исполняется команда `next(action)`, которая передаст это действие любому другому коду, проявившему к нему интерес.

Новое действие `SEARCH_RESULTS` можно создавать при отправке сетевого вызова, получении данных или обнаружении ошибок (листинг 5.17).

Листинг 5.17. Код действия типа SEARCH_RESULTS

```
store.dispatch({
  type: 'SEARCH_RESULTS',
```

```
  payload: {
    loading: ...,
    error: ...,
    data: ...
  },
})
```

Наше новое действие содержит следующую полезную нагрузку:

- булеву переменную `loading`, значение которой равно `true`, пока исполняется сетевой запрос;
- объект `data`, содержащий ответ от сервера;
- объект `error`, содержащий информацию о любой случившейся ошибке[6].

Теперь можно создать преобразователь, который сохранит действие `SEARCH_RESULTS` в общем состоянии (листинг 5.18).

Листинг 5.18. Преобразователь для сохранения действия SEARCH_RESULTS

```
const reducer = (state, action) => {
  if (action.type === 'SEARCH_RESULTS') {
    return {
      ...state,
      searchResults: { ...action.payload },
    }
  }
  return { ...state }
}
export default reducer
```

Нам также нужно зарегистрировать наше промежуточное ПО при создании хранилища Redux, используя для этого функцию Redux `applyMiddleware`. В нашем примере это делается в файле App.js (листинг 5.10).

Листинг 5.19. Регистрация промежуточного ПО

```
import { Provider } from 'react-redux'
import { createStore, applyMiddleware } from 'redux'
import './App.css'
import reducer from './reducer'
import Search from './Search'
import axiosMiddleware from './axiosMiddleware'
const store = createStore(reducer, applyMiddleware(axiosMiddleware))
```

[6] Для простоты мы просто сохраняем весь объект. В настоящем приложении следует обеспечить, чтобы объект `error` содержал только сериализуемый текст.

```
function App() {
   return (
      <div className="App">
         <Provider store={store}>
            <Search />
         </Provider>
      </div>
   )
}
export default App
```

Наконец, мы можем собрать все в компоненте `Search`, который будет отправлять запрос поиска, а затем считывать результаты посредством селектора Redux (листинг 5.20).

Листинг 5.20. Объединение всех составляющих приложения в компоненте `Search`

```
import './App.css'
import { useState } from 'react'
import { useDispatch, useSelector } from 'react-redux'
const Search = () => {
   const [terms, setTerms] = useState()
   const {
      data: results,
      error,
      loading,
   } = useSelector((state) => state.searchResults || {})
   const dispatch = useDispatch()
   return (
      <div className="App">
         <input
            placeholder="Search..."
            type="text"
            value={terms}
            onChange={(e) => {
               setTerms(e.target.value)
               dispatch({
                  type: 'SEARCH',
                  payload: e.target.value,
               })
            }}
         />
         {error ? (
            <p>Error: {error.message}</p>
         ) : loading ? (
            <p>Loading...</p>
```

```
        ) : results && results.length ? (
          <table>
            <thead>
              <tr>
                <th>City</th>
                <th>State</th>
              </tr>
            </thead>
            {results.map((r) => (
              <tr>
                <td>{r.name}</td>
                <td>{r.state}</td>
              </tr>
            ))}
          </table>
        ) : (
          <p>No results</p>
        )}
      </div>
  )
}
export default Search
```

На рис. 5.10 показаны разные стадии исполнения этого приложения.

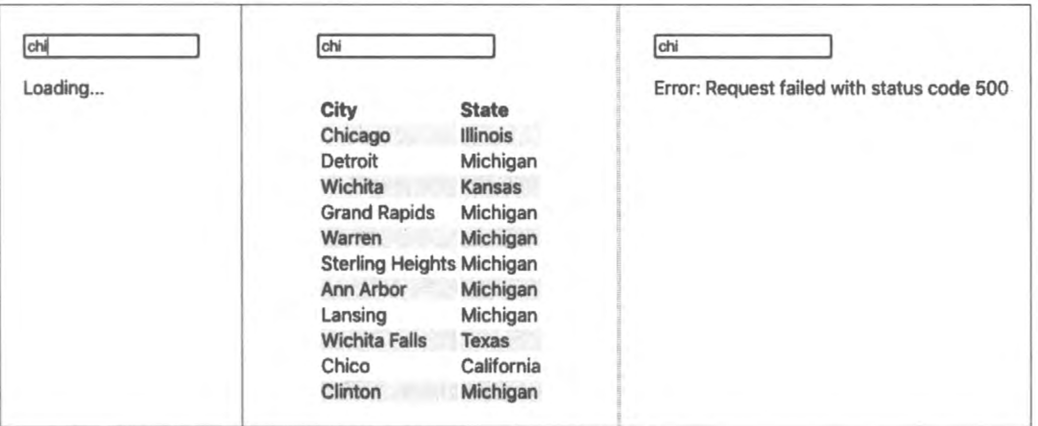


Рис. 5.10. Окно приложения при загрузке данных (слева), с загруженными данными (посередине) и при ошибке (справа)

Обсуждение

Преобразователи Redux всегда обрабатывают действия в строгом порядке их отправки. То же самое относится и к сетевым запросам, генерируемым промежуточным ПО. Но если создавать большое количество сетевых запросов в быстрой

последовательности друг за другом, то может оказаться, что ответы на них возвратятся в другом порядке. Если это обстоятельство повышает вероятность возникновения ошибок, то следует рассмотреть возможность использования маркеров отмены[7].

Также может быть желательным рассмотреть возможность перемещения всего кода Redux `useDispatch()/useSelector()` из компонентов в специализированные хуки, что даст более гибкую архитектуру вследствие отделения служебного уровня от кода компонентов.

Исходный код данного рецепта можно загрузить на веб-сайте GitHub по адресу **https://oreil.ly/YlqEF**.

5.5. Подключение к GraphQL

ЗАДАЧА

Язык запросов GraphQL замечательно подходит для создания API-интерфейсов. Для тех из вас, кто в течение некоторого времени пользовался службами REST, некоторые возможности GraphQL будут выглядеть странными (или даже святотатствующими), но, поработав на нескольких проектах с использованием GraphQL, мы, безусловно, можем порекомендовать вам рассмотреть возможность его применения в своем следующем разрабатываемом проекте.

Когда говорят о GraphQL, то могут иметь в виду разные вещи. Это может быть язык GraphQL, управляемый и поддерживаемый организацией GraphQL Foundation. Язык GraphQL предназначен для указания API-интерфейсов и создания запросов для доступа и изменения данных, хранящихся за этими интерфейсами. Это также может быть сервер GraphQL, который объединяет несколько низкоуровневых методов доступа к данным в полноценную веб-службу. Наконец, это может быть клиент GraphQL, который позволяет быстро создавать клиентские запросы при очень небольшом объеме кода и передавать по сети только требуемые данные.

Но как можно интегрировать возможности GraphQL в приложения React?

РЕШЕНИЕ

Прежде чем рассматривать использование GraphQL в React, создадим простой сервер GraphQL. Первое, что нам нужно, — это схема GraphQL. Схема — это формальное определение данных и служб, которые будет предоставлять наш сервер GraphQL.

В листинге 5.21 приведена схема `schema.graphql`, которую мы будем применять в нашем сервере. Это спецификация GraphQL примера форумного сообщения, рассмотренного ранее в этой главе.

---

[7] См. *раздел 5.3.*

Листинг 5.21. Схема schema.graphql для форумного сообщения

```
type Query {
  messages: [Message]
}
type Message {
  id: ID!
  author: String!
  text: String!
}

type Mutation {
  addMessage(
    author: String!
    text: String!
  ): Message
}
```

Данная схема определяет один *запрос* (метод для чтения данных), называющийся `messages`, который возвращает набор объектов типа `Message`. Каждый объект `Message` имеет идентификатор `author` с ненулевым значением строчного типа и ненулевую строку `text`. В объекте также указана одна *мутация* (метод для изменения данных), называющаяся `addMessage`, которая будет сохранять сообщение на основании строки `author` и строки `string`.

Прежде чем приступать к созданию нашего сервера, установим требуемые фреймворки и библиотеки:

```
$ npm install apollo-server
$ npm install graphql
$ npm install require-text
```

Фреймворк `appolo-server` предназначен для создания серверов GraphQL. А библиотека `require-text` служит для чтения файла схемы `schema.graphql`. Теперь можно создать наш сервер. Его код из файла server.js приведен в листинге 5.22.

Листинг 5.22. Код сервера GraphQL

```
const { ApolloServer } = require('apollo-server')
const requireText = require('require-text')
const typeDefs = requireText('./schema.graphql', require)
const messages = [
  {
    id: 0,
    author: 'SC',
    text: 'Rolls complete and a pitch is program. One BRAVO.',
  },
```

```
  {
    id: 1,
    author: 'PAO',
    text: 'One BRAVO is an abort control model. Altitude is 2 miles.',
  },
  {
    id: 2,
    author: 'CAPCOM',
    text: 'All is well at Houston. You are good at 1 minute.',
  },
]
const resolvers = {
  Query: {
    messages: () => messages,
  },
  Mutation: {
    addMessage: (parent, message) => {
      const item = { id: messages.length + 1, ...message }
      messages.push(item)
      return item
    },
  },
}
const server = new ApolloServer({
  typeDefs,
  resolvers,
})
server.listen({ port: 5000 }).then(({ url }) => {
  console.log(Launched at ${url}!)
})
```

Сервер сохраняет сообщения в массиве, который предварительно заполнен несколькими сообщениями. Сервер запускается на исполнение следующей командой:

```
$ node ./server.js
```

Сервер работает по адресу **http://localhost:5000**. В браузере по этому адресу открывается окно клиента GraphQL Playground (рис. 5.11). Клиент Playground позволяет тестировать запросы и мутации, исполняя их в интерактивном режиме, прежде чем использовать их в рабочем коде.

Теперь можно приступать к рассмотрению кода клиента React. Но сначала установим клиент Apollo:

```
$ npm install @apollo/client
```

Сервер GraphQL поддерживает как GET- так и POST-запросы, но клиент Apollo отправляет запросы и мутации в виде запросов POST, что позволяет избежать междоменных проблем, а также подключаться к сторонним GraphQL-серверам, не прибе-

гая к использованию прокси-сервера. Вследствие этого клиент GraphQL должен самостоятельно обеспечивать функции кеширования. Поэтому нам нужно задать кеш и адрес сервера в настройках клиента в файле App.js (листинг 5.23).

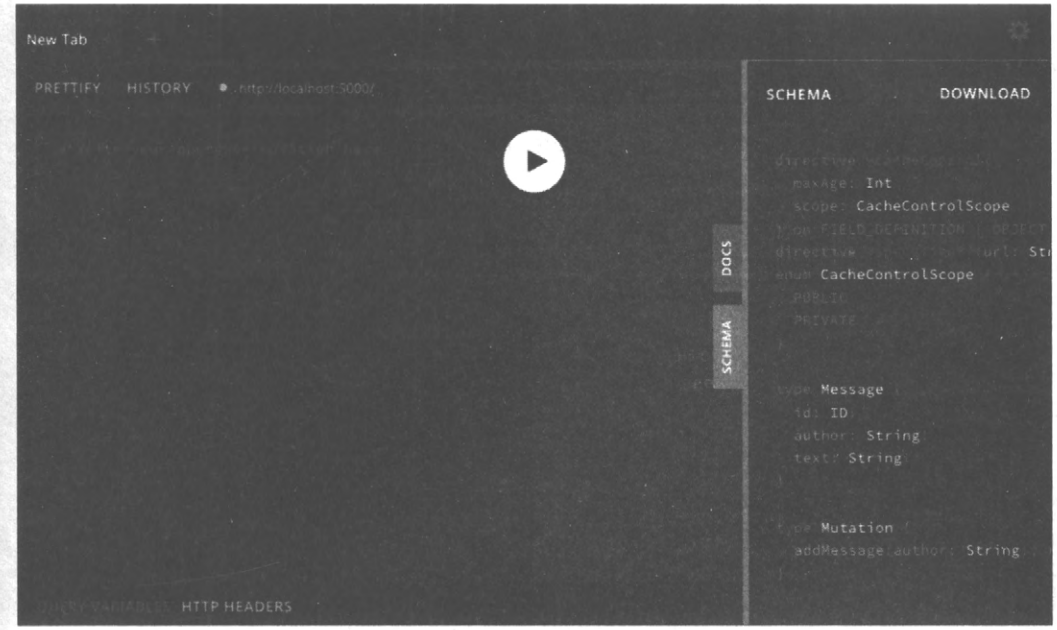


Рис. 5.11. Клиент GraphQL Playground исполняется по адресу **http://localhost:5000**

Листинг 5.23. Конфигурирование кеша и адреса сервера

```
import './App.css'
import {
  ApolloClient,
  ApolloProvider,
  InMemoryCache,
} from '@apollo/client'
import Forum from './Forum'

const client = new ApolloClient({
  uri: 'http://localhost:5000',
  cache: new InMemoryCache(),
})
function App() {
  return (
    <div className="App">
      <ApolloProvider client={client}>
        <Forum />
      </ApolloProvider>
```

```
      </div>
   )
}
export default App
```

Компонент AppoloProvider обеспечивает доступность клиента любому дочернему компоненту. Если не включить этот компонент в список импортирования, то весь клиентский код GraphQL не будет работать.

Вызовы к серверу GraphQL будем осуществлять из компонента Forum. Для этого будем выполнять два действия:

- запрос Messages, считывающий все сообщения;
- мутацию AddMessages, отправляющую новое сообщение.

Запрос и мутация составлены на языке GraphQL. Содержимое запроса Messages приведено в листинге 5.24.

Листинг 5.24. Содержимое запроса Messages

```
query Messages {
   messages {
      author text
   }
}
```

Данный запрос означает, что мы хотим прочитать все сообщения, но возвратить только строки author и text. Идентификатор id сообщения не будет возвращен, поскольку мы не запрашиваем его. Это одна из составляющих гибкости GraphQL: требуемые элементы указываются в запросе, а не в отдельном вызове API для каждой разновидности.

Мутация AddMessage чуть посложнее, поскольку ее нужно параметризировать, чтобы можно было указывать значения для author и text при каждом ее вызове (листинг 5.25).

Листинг 5.25. Код мутации AddMessage

```
mutation AddMessage(
   $author: String!
   $text: String!
) {
   addMessage(
      author: $author
      text: $text
) {
      author
      text
   }
}
```

Мы будем использовать хуки `useQuery` и `useMutation`, предоставляемые клиентом Apollo среды GraphQL. Хук `useQuery` возвращает объект с атрибутами `data`, `loading` и `error`[8]. А хук `useMutation` возвращает массив с двумя значениями: функцией и объектом, представляющим результат.

В *разделе 5.2* мы рассмотрели, как автоматически обновлять данные после того, как они были изменены на сервере какой-либо мутацией. К счастью, клиент Apollo оснащен готовым механизмом решения этой задачи. В частности, при вызове мутации можно указать массив других запросов, который нужно возвратить в случае успешного выполнения мутации (листинг 5.26).

Листинг 5.26. Вызов мутации `addMessage` с указанием массива другого запроса

```
await addMessage({
   variables: { author, text },
   refetchQueries: ['Messages'],
})
```

Строка `Messages` представляет название запроса GraphQL, означая, что мы можем исполнять множественные запросы к службе GraphQL и указывать, который из них может требовать обновления после изменений.

Наконец, у нас есть все составляющие компонента `Forum`, полный код которого приведен в листинге 5.27.

Листинг 5.27. Полный код компонента `Forum`

```
import { gql, useMutation, useQuery } from '@apollo/client'
import { useState } from 'react'
const MESSAGES = gql`
   query Messages {
      messages {
         author
         text
      }
   }
const ADD_MESSAGE = gql`
   mutation AddMessage($author: String!, $text: String!) {
      addMessage(author: $author, text: $text) {
      author
      text
   }
}
`
```

[8] Это стандартный набор значений для асинхронной службы. Мы использовали их в других рецептах в этой главе.

```
const Forum = () => {
   const {
      loading: messagesLoading,
      error: messagesError,
      data,
   } = useQuery(MESSAGES)
const [addMessage] = useMutation(ADD_MESSAGE)
const [text, setText] = useState()
const [author, setAuthor] = useState()

const messages = data && data.messages
return (
   <div className="App">
      <input
         type="text"
         value={author}
         placeholder="Author"
         onChange={(evt) => setAuthor(evt.target.value)}
      />
      <textarea
         value={text}
         placeholder="Message"
         onChange={(evt) => setText(evt.target.value)}
      />
      <button
         onClick={async () => {
            try {
               await addMessage({
                  variables: { author, text },
                  refetchQueries: ['Messages'],
               })
               setText('')
               setAuthor('')
            } catch (err) {}
         }}
      >
         Post
      </button>
      {messagesError ? (
         <div className="error">
            Something went wrong:
            <div className="error-contents">
               {messagesError.message}
            </div>
         </div>
      ) : messagesLoading ? (
         <div className="loading">Loading...</div>
```

```
      ) : messages && messages.length ? (
        <dl>
          {messages.map((m) => (
            <>
              <dt>{m.author}</dt>
              <dd>{m.text}</dd>
            </>
          ))}
        </dl>
        ) : (
          'No messages'
        )}
      </div>
    )
}
export default Forum
```

Теперь отправленное сообщение автоматически добавляется в конец списка сообщений, как показано на рис. 5.12.

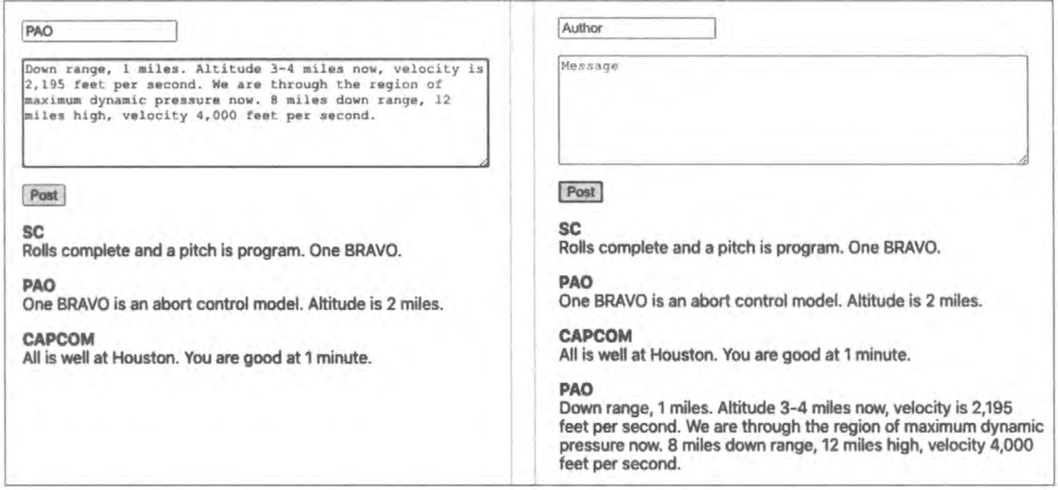


Рис. 5.12. Отправленные сообщения автоматически добавляются в конец списка

Обсуждение

Платформа GraphQL особенно полезна, когда группа разделена на разработчиков бэкенда и разработчиков фронтенда. В отличие от платформы REST, система GraphQL не требует, чтобы разработчики бэкенда предоставляли специализированную обработку для каждого клиентского вызова API. Вместо этого они могут просто предоставить монолитную и единообразную структуру API, оставив команде фронтенда решать, как именно они будут с ней работать.

При создании приложений React на основе GraphQL может быть полезным рассмотреть возможность переместить все вызовы хуков в `useQuery` и `useMutation` в специализированные хуки[9]. Таким образом вы создадите более гибкую архитектуру, в которой компоненты не так сильно привязаны к подробностям служебного уровня.

Исходный код данного рецепта можно загрузить на веб-сайте GitHub по адресу **https://oreil.ly/xTcAK**.

5.6. Уменьшение сетевой нагрузки при помощи очищенных запросов

ЗАДАЧА

Работая в системе разработки, легко забыть об эффективности приложения. Возможно, это и к лучшему, поскольку более важно, чтобы код работал правильно, нежели быстро, но при этом неправильно.

Но когда приложение попадает в реальную обстановку, например в среду для тестирования на приемлемость для пользователя, тогда вопрос производительности становится более важным. Ассоциированные с React динамические интерфейсы часто осуществляют множество сетевых вызовов, и стоимость этих вызовов будет заметна только тогда, когда серверу необходимо одновременно обслуживать большое число клиентов.

В этой главе мы несколько раз работали с приложением поиска. В этом приложении пользователь может искать город по его имени или по названию штата, в котором он находится. Поиск выполняется моментально, пока пользователь вводит данные. Если в браузере открыть окно **Инструменты веб-разработчика**, а в нем выбрать вкладку **Сеть**, то мы увидим, что приложение создает сетевой запрос при вводе каждого символа искомой строки (рис. 5.13).

Большинство из этих сетевых запросов не предоставляют почти никаких полезных данных. Средний пользователь, наверное, вводит один символ каждые полсекунды,

Name	Status	Type
search?terms=c	200	xhr
search?terms=ch	200	xhr
search?terms=chi	200	xhr
search?terms=chic	200	xhr
search?terms=chica	200	xhr
search?terms=chicag	200	xhr
search?terms=chicago	200	xhr

Рис. 5.13. Приложение поиска исполняет сетевой запрос при вводе каждого символа

[9] Во многом подобно обработке сетевых вызовов HTTP в *разделе 5.2*.

и если при этом он смотрит на клавиатуру, то, скорее всего, он даже не увидит результатов поиска, возвращаемых после каждого символа. Из семи запросов данного примера, отправленных серверу, он, скорее всего, прочитает результаты только одного: последнего. Это означает, что сервер выполняет в семь раз больший объем работы, чем необходимо.

Что можно сделать, чтобы не отправлять столь много напрасных запросов?

РЕШЕНИЕ

Для решения этой задачи мы очистим сетевые запросы от лишних вызовов поиска. Под очисткой сетевых запросов имеется в виду задержка отправки сетевого запроса на очень короткое время, скажем полсекунды. Если в течение этого периода ожидания поступает другой запрос, мы отбрасываем первый запрос и опять задерживаем отправку нового запроса, и т. д. Таким образом, мы задерживаем отправку всех запросов до тех пор, пока не будем получать новых запросов в течение полсекунды.

Чтобы разобраться, как это работает, рассмотрим код хука поиска нашего примера приложения из файла useSearch.js (листинг 5.28).

Листинг 5.28. Код хука useSearch

```
import { useEffect, useState } from 'react'
import axios from 'axios'
const useSearch = (terms) => {
   const [data, setData] = useState([])
   const [loading, setLoading] = useState(false)
   const [error, setError] = useState()

   useEffect(() => {
      let didCancel = false
      setError(null)
      if (terms) {
         ;(async () => {
            try {
               setLoading(true)
               const response = await axios.get('/search', {
                  params: { terms },
               })
               if (!didCancel) {
                  setData(response.data)
               }
            } catch (err) {
               setError(err)
            } finally {
               setLoading(false)
            }
         })()
```

```
    } else {
      setData([])
      setLoading(false)
    }
    return () => {
      didCancel = true
    }
  }, [terms])
  return { data, loading, error }
}
export default useSearch
```

Код, который отправляет сетевой запрос, находится в блоке `(async ()....)()`. Нам нужно задержать исполнение этого кода, пока не получим полсекунды без новых запросов.

После этой задержки код запустится на исполнение функцией JavaScript `setTimeout`. Это будет ключевым моментом реализации возможности очистки:

```
const newTimer = setTimeout(SOMEFUNCTION, 500)
```

Новое значение `newTimer` можно учесть, чтобы очистить счетчик тайм-аута. Если сделать это достаточно быстро, то вызов функции никогда не будет осуществлен. Чтобы разобраться, как с помощью этого кода реализовать очистку сетевых запросов, рассмотрим код файла useSearch.js, модифицированный кодом очистки и теперь хранящийся в файле useDebouncedSearch.js (листинг 5.29).

Листинг 5.29. Код файла useDebouncedSearch.js

```
import { useEffect, useState } from 'react'
import axios from 'axios'
const useDebouncedSearch = (terms) => {
  const [data, setData] = useState([])
  const [loading, setLoading] = useState(false)
  const [error, setError] = useState()
  useEffect(() => {
    setError(null)
    if (terms) {
      const newTimer = setTimeout(() => {
        ;(async () => {
          try {
            setLoading(true)
            const response = await axios.get('/search', {
              params: { terms },
            })
            setData(response.data)
          } catch (err) {
            setError(err)
```

```
            } finally {
                setLoading(false)
            }
        })()
    }, 500)
    return () => clearTimeout(newTimer)
  } else {
    setData([])
    setLoading(false)
  }
 }, [terms])
 return { data, loading, error }
}
export default useDebouncedSearch
```

Мы передаем код сетевых операций функции `setTimeout`, а затем возвращаем следующее:

```
() => clearTimeout(newTimer)
```

Если возвращать функцию из хука `useEffect`, то этот код вызывается непосредственно перед следующим запуском данного хука. Если пользователь вводит символы довольно быстро, то отправка сетевого запроса будет откладываться. Только при отсутствии ввода от пользователя в течение свыше полсекунды этот код отправит сетевой запрос.

Исходная версия этого хука, `useSearch`, отправляла сетевой запрос для каждого вводимого символа. А в модифицированном варианте с кодом очистки, `useDebouncedSearch`, ввод искомой строки со средней скоростью вызовет только один сетевой запрос (рис. 5.14).

Name	Status	Type
☐ search?terms=chicago	200	xhr

Рис. 5.14. Модифицированный вариант хука поиска будет отправлять меньше запросов

Обсуждение

Очистка сетевых запросов от избыточных вызовов поиска уменьшит объем сетевого трафика и нагрузку на сервер. При этом важно помнить, что очистка запросов только уменьшает число ненужных сетевых запросов, но не устраняет проблему возврата ответов на запросы в неправильном порядке. Подробно решение проблемы возврата ответов в неправильном порядке рассматривается в *разделе 5.3*.

Исходный код данного рецепта можно загрузить на веб-сайте GitHub по адресу **https://oreil.ly/5nciD**.

ГЛАВА 6

Библиотеки компонентов

Для разработки приложения любого размера, скорее всего, потребуется библиотека компонентов. Круг типов данных, поддерживаемых нативным HTML, несколько ограничен, а их реализации могут варьироваться в зависимости от браузера. Например, поле для ввода даты выглядит очень по-разному в браузерах Chrome, Firefox и Edge.

Библиотеки компонентов позволяют создать единообразный внешний вид для разрабатываемого приложения. Они также часто хорошо подстраиваются при переключении между настольными и мобильными клиентами. Самое важное, библиотеки компонентов часто повышают удобство в использовании приложения. Такие библиотеки или создаются согласно всесторонне протестированным проектно-конструкторским стандартам (например, библиотека Material Design), или разрабатываются в течение нескольких лет. Поэтому все проблемные аспекты обычно сглажены.

Но при этом следует иметь в виду, что идеальных библиотек компонентов не существует. Все они имеют свои сильные и слабые стороны, и нужно выбирать такую библиотеку, которая лучше всего отвечает вашим требованиям. При наличии большой UX-команды разработчиков и робастного набора существующих стандартов проектирования вам, скорее всего, будет нужна библиотека, позволяющая выполнять большое количество подстроек, чтобы соответствовать вашим корпоративным темам. Примером такой библиотеки будет библиотека Material-UI, которая позволяет довольно значительно модифицировать свои компоненты. В случае же небольшой UX-команды или вообще при ее отсутствии вам, скорее всего, подойдет библиотека наподобие удобной и функциональной библиотеки Semantic UI, которая позволит вам подготовиться и приступить к работе в быстром порядке.

Но независимо от выбранной библиотеки всегда помните, что важным аспектом проектирования и разработки пользовательских интерфейсов приложений является не внешний вид приложения, а его поведение. Пользователи вскоре перестанут обращать внимание на всю крикливую графику, навешанную на интерфейс, но они никогда не забудут, а главное, не простят, если какая-либо часть интерфейса будет вызывать у них раздражение при каждом использовании.

6.1. Использование библиотеки Material Design совместно с библиотекой Material-UI

ЗАДАЧА

В настоящее время многие приложения предназначены для исполнения как в Сети, так и в виде нативного ПО на мобильных устройствах. Компания Google разработала спецификацию Material Design для обеспечения одинакового восприятия приложений на всех платформах. Элементы этой спецификации будут выглядеть знакомыми вашим пользователям, если они также пользуются смартфонами на Android или любыми иными продуктами разработки Google. Существует несколько реализаций этой спецификации, одной из которых является библиотека Material-UI для платформы React. Но как нам установить эту библиотеку и как потом работать с ней?

РЕШЕНИЕ

Библиотека Material-UI устанавливается следующим образом:

```
$ npm install @material-ui/core
```

Базовая библиотека содержит основные компоненты, но в ней отсутствует одна важная возможность — стандартный шрифт. Чтобы придать такое же восприятие компонентов библиотеки, как в нативных мобильных приложениях, нужно установить шрифт Roboto компании Google:

```
$ npm install fontsource-roboto
```

В спецификации Material Design также определен большой набор стандартных значков. Эти значки предоставляют общий визуальный язык для таких стандартных задач, как редактирование, создание новых элементов, коллективное использование содержимого и т. п. Чтобы получить высококачественные версии этих значков, нужно также установить библиотеку значков Material-UI:

```
$ npm install @material-ui/icons
```

Теперь, когда у нас все готово для использования библиотеки Material-UI, выясним, что можно делать с ее помощью. Здесь мы не можем описать подробно все доступные компоненты библиотеки, поэтому рассмотрим только некоторые наиболее популярные возможности[1].

Начнем с исследования основ возможностей стилистического оформления, предоставляемых библиотекой Material-UI. Чтобы обеспечить единообразный вид компонентов Material-UI в разных браузерах, библиотека содержит компонент `CssBaseline`, который нормализует базовое стилистическое оформление разрабатываемого приложения, удаляет отступы и применяет стандартные фоновые цвета. Этот компо-

---

[1] Подробная информация обо всем наборе компонентов предоставляется на веб-сайте Material-UA (**https://material-ui.com**).

нент следует вставить в разрабатываемое приложение где-то в самом начале. Например, в приложениях `create-react-app` его следует добавить в файл App.js, как показано в листинге 6.1.

Листинг 6.1. Добавление компонента `CssBaseling` в файл App.js

```
import CssBaseline from '@material-ui/core/CssBaseline'
...
function App() {
   // ...
   return (
      <div className="App">
         <CssBaseline />
         ...
      </div>
   )
}
export default App
```

Далее, взглянем на компоненты `AppBar` и `Toolbar`. Эти компоненты предоставляют стандартные заголовки, которые можно увидеть в большинстве приложений с использованием Material Design и в которых будут отображаться другие элементы, такие как меню "гамбургер" и панели меню.

Мы поместим компонент `AppBar` вверху экрана, а внутрь его вставим компонент `Toolbar` (листинг 6.2). Это даст нам возможность посмотреть, как обрабатывается печать внутри Material-UI.

Листинг 6.2. Добавление компонентов `AppBar` и `Toolbar`

```
<div className="App">
   <CssBaseline/>
   <AppBar position='relative'>
      <Toolbar>
         <Typography component='h1' variant='h6' color='inherit' noWrap>
            Material-UI Gallery
         </Typography>
      </Toolbar>
   </AppBar>
   <main>
      {/* Сюда вставляется основное содержимое...*/}
   </main>
</div>
```

Хотя в приложениях с использованием Material-UI можно вставлять обычное текстовое содержимое, в целом будет лучше отображать его внутри компонента `Typography`, который обеспечит совместимость текста со стандартами Material Design.

Этот же компонент пригоден для отображения текста внутри соответствующих элементов разметки. В данном случае мы будем отображать текст в компоненте `Toolbar` в формате элемента `h1`. Именно таково определение атрибута `component` компонента `Typography`: элемент HTML, который должен применяться для заключения в нем текста. Но мы также можем дать указание библиотеке Material-UI придать тексту стилистическое оформление, например в виде заголовка `h6`. Это уменьшит размер текста и сделает заголовок менее бросающимся в глаза.

Далее рассмотрим, каким образом посредством возможностей библиотеки Material-UI осуществляется стилистическое оформление вывода. Для этого предназначены темы. Тема — это объект JavaScript, который определяет иерархию стилей CSS. Темы можно определять централизованно, что позволяет управлять общим внешним видом разрабатываемого приложения.

Темы по умолчанию можно модифицировать. Но чтобы получить возможность создавать модифицированные версии тем по умолчанию, нужно импортировать функцию `makeStyles`:

```
import { makeStyles } from '@material-ui/core/styles'
```

Наше приложение будет отображать галерею изображений, поэтому нам нужно создать стили для элементов галереи, описаний и т. п. Стили для всех этих разных экранных элементов мы можем создать при помощи функции `makeStyles`, как показано в листинге 6.3.

Листинг 6.3. Создание стилей при помощи функции makeStyles

```
const useStyles = makeStyles((theme) => ({
  galleryGrid: {
    paddingTop: theme.spacing(4),
  },
  galleryItemDescription: {
    overflow: 'hidden',
    textOverflow: 'ellipsis',
    whiteSpace: 'nowrap',
  },
}))
```

В этом упрощенном примере мы расширяем базовую тему, включая в нее стили для классов `galleryGrid` и `galleryItemDescription`. Атрибуты CSS можно добавлять напрямую или (в случае атрибута `paddingTop` в классе `galleryGrid`) можно ссылаться на какое-либо значение в текущей теме (в данном случае это `theme.spacing(4)`). Таким образом, появляется возможность перенаправить части стилистического оформления к центральной теме, где мы сможем изменить их в дальнейшем.

Хук `useStyles`, возвращаемый функцией `makeStyles`, создает набор классов CSS, а затем возвращает их названия, чтобы к ним можно было обращаться изнутри разрабатываемого компонента.

Например, нам нужно будет отобразить матрицу изображений, используя компоненты Container и Grid[2]. Для этого мы можем прикрепить к ним стили из темы, как показано в листинге 6.4.

Листинг 6.4. Прикрепление стилей из темы к компонентам

```
const classes = useStyles()
return (
  <div className="App">
    ...
    <main>
      <Container className={classes.galleryGrid}>
        <Grid container spacing="4">
          <Grid item>...</Grid>
          <Grid item>...</Grid>
          ...
        </Grid>
      </Container>
    </main>
  </div>
)
```

Каждый компонент Grid является или *контейнером*, или *элементом*. Мы будем отображать одно изображение галереи в каждом элементе.

Согласно политике Material Design важные элементы отображаются в *карточках*. Карточка представляет собой прямоугольную панель, которая кажется плавающей на небольшой высоте над фоном. Если вам когда-либо приходилось использовать магазин Google Play, то вы должны были видеть карточки, в которых отображаются приложения, музыкальные дорожки или другие вещи, предлагаемые для загрузки. Мы поместим по карточке в каждый элемент Grid и будем применять ее для предварительного просмотра изображения, отображения описания изображения, а также кнопки для вывода более подробной версии изображения. На рис. 6.1 показано такое использование карточек в разрабатываемом приложении.

Библиотека Material-UI также предоставляет обширную поддержку для диалоговых окон. В листинге 6.5 приведен пример кода для пользовательского диалогового окна.

Листинг 6.5. Пример кода для пользовательского диалогового окна

```
import Dialog from '@material-ui/core/Dialog'
import DialogTitle from '@material-ui/core/DialogTitle'
import Typography from '@material-ui/core/Typography'
import DialogContent from '@material-ui/core/DialogContent'
import DialogActions from '@material-ui/core/DialogActions'
import Button from '@material-ui/core/Button'
```

[2] Дополнительная информация по этим компонентам предоставляется на веб-сайте библиотеки Material-UI site (**https://material-ui.com**).

```
import CloseIcon from '@material-ui/icons/Close'
const MyDialog = ({ onClose, open, title, children }) => {
   return (
      <Dialog open={open} onClose={onClose}>
         <DialogTitle>
            <Typography
               component="h1"
               variant="h5"
               color="inherit"
               noWrap
            >
               {title}
            </Typography>
         </DialogTitle>
         <DialogContent>{children}</DialogContent>
         <DialogActions>
            <Button
               variant="outlined"
               startIcon={<CloseIcon />}
               onClick={onClose}
            >
               Close
            </Button>
         </DialogActions>
      </Dialog>
   )
}
export default MyDialog
```

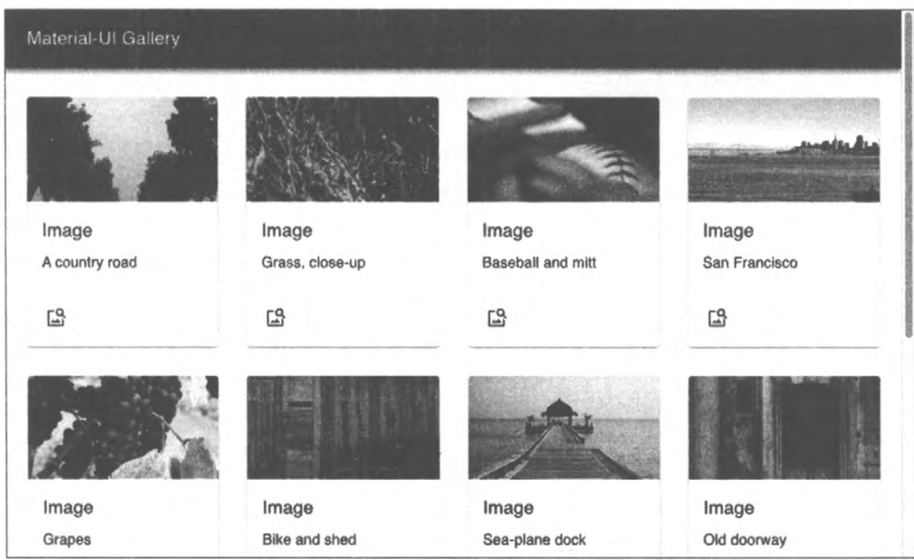


Рис. 6.1. Карточки находятся внутри элементов матрицы Grid, которые, в свою очередь, находятся внутри контейнеров

Обратите внимание на то, что мы импортируем значок SVG из коллекции значков библиотеки Material-UI, которую установили ранее. Вверху диалогового окна находится его название `DialogTitle`. Действия `DialogActions` представляют кнопки, которые отображаются внизу диалогового окна. Основное тело диалогового окна определяется в `DialogContent`.

В листинге 6.6 приведен полный код файла App.js приложения.

Листинг 6.6. Полный код файла App.js приложения

```
import './App.css'
import CssBaseline from '@material-ui/core/CssBaseline'
import AppBar from '@material-ui/core/AppBar'
import { Toolbar } from '@material-ui/core'
import Container from '@material-ui/core/Container'
import Grid from '@material-ui/core/Grid'
import Card from '@material-ui/core/Card'
import CardMedia from '@material-ui/core/CardMedia'
import CardContent from '@material-ui/core/CardContent'
import CardActions from '@material-ui/core/CardActions'
import Typography from '@material-ui/core/Typography'
import { makeStyles } from '@material-ui/core/styles'
import { useState } from 'react'
import MyDialog from './MyDialog'
import ImageSearchIcon from '@material-ui/icons/ImageSearch'
import gallery from './gallery.json'
import IconButton from '@material-ui/core/IconButton'
const useStyles = makeStyles((theme) => ({
   galleryGrid: {
      paddingTop: theme.spacing(4),
   },
   galleryItem: {
      height: '100%',
      display: 'flex',
      flexDirection: 'column',
      // maxWidth: '200px'
   },
   galleryImage: {
      paddingTop: '54%',
   },
   galleryItemDescription: {
      overflow: 'hidden',
      textOverflow: 'ellipsis',
      whiteSpace: 'nowrap',
   },
}))
```

```
function App() {
   const [showDetails, setShowDetails] = useState(false)
   const [selectedImage, setSelectedImage] = useState()
   const classes = useStyles()
   return (
      <div className="App">
         <CssBaseline />
         <AppBar position="relative">
            <Toolbar>
               <Typography
                  component="h1"
                  variant="h6"
                  color="inherit"
                  noWrap
               >
                  Material-UI Gallery
               </Typography>
            </Toolbar>
         </AppBar>
         <main>
            <Container className={classes.galleryGrid}>
               <Grid container spacing="4">
                  {gallery.map((item, i) => {
                   return (
                    <Grid item key={`photo-${i}`} xs={12} sm={3} lg={2}>
                           <Card className={classes.galleryItem}>
                              <CardMedia
                                 image={item.image}
                                 className={classes.galleryImage}
                                 title="A photo"
                              />
                              <CardContent>
                                 <Typography
                                    gutterBottom
                                    variant="h6"
                                    component="h2"
                                 >
                                    Image
                                 </Typography>
                                 <Typography
                                className={classes.galleryItemDescription}
                                 >
                                    {item.description}
                                 </Typography>
                              </CardContent>
                              <CardActions>
                                 <IconButton
                                    aria-label="delete"
```

```
                        onClick={() => {
                          setSelectedImage(item)
                          setShowDetails(true)
                        }}
                        color="primary"
                      >
                        <ImageSearchIcon />
                      </IconButton>
                    </CardActions>
                  </Card>
                </Grid>
              )
            }))}
          </Grid>
        </Container>
      </main>
      <MyDialog
        open={showDetails}
        title="Details"
        onClose={() => setShowDetails(false)}
      >
        <img
          src={selectedImage && selectedImage.image}
          alt="From PicSum"
        />
          <Typography>
            {selectedImage && selectedImage.description}
          </Typography>
        </MyDialog>
      </div>
    )
}
export default App
```

Обсуждение

Библиотека Material-UI замечательно подходит для разработки веб-приложений; это одна из наиболее популярных библиотек, существующая в настоящее время для разработки приложений React. Посетители вашего приложения почти наверняка сталкивались с оформлением средствами этой библиотеки в других приложениях, что повысит его удобство в использовании. Но прежде, чем приступать к работе с библиотекой Material-UI в своем приложении, будет полезным уделить некоторое время изучению принципов платформы Material Design (**https://oreil.ly/Jlk7w**). Таким образом, ваше приложение будет не только привлекательным, но также удобным в использовании и доступным (**https://oreil.ly/RJiW1**) для пользователей.

Исходный код данного рецепта можно загрузить на веб-сайте GitHub по адресу **https://oreil.ly/TqVFz**.

6.2. Простой пользовательский интерфейс посредством React Bootstrap

ЗАДАЧА

Наиболее популярная библиотека CSS в последние 10 лет — это, наверное, библиотека Bootstrap от компании Twitter. Данная библиотека будет также хорошим выбором, когда при разработке нового приложения у вас нет лишнего времени, чтобы создавать специализированный пользовательский интерфейс, и вы просто хотите взять какой-либо интерфейс, который был бы удобным и знакомым многим пользователям.

Но библиотека Bootstrap имеет свои истоки в эпохе, когда фреймворков наподобие React не существовало. Эта библиотека содержит ресурсы CSS и набор библиотек JavaScript, предназначенных для работы с веб-страницами, содержащими небольшой объем специализированного клиентского кода. Функциональность базовой библиотеки Bootstrap не очень хорошо вписывается в совместную работу с фрейворками наподобие React.

Так как же оснастить возможностями Bootstrap разрабатываемые нами приложения на основе React?

РЕШЕНИЕ

Существует несколько модификаций библиотеки Bootstrap для работы с React. В этом рецепте мы рассмотрим одну из них — библиотеку React Bootstrap. Эта библиотека работает совместно со стандартными библиотеками CSS Bootstrap, расширяя код JavaScript библиотеки Bootstrap, чтобы улучшить совместимость с React.

Для начала установим компоненты React Bootstrap и библиотеки JavaScript Bootstrap:

```
$ npm install react-bootstrap bootstrap
```

Библиотека React Bootstrap не содержит никаких собственных таблиц стилей CSS, поэтому вам нужно будет самому включить их копию. Стандартный способ для этого — загрузить требуемый файл из сети доставки контента (CDN) в коде HTML. Например, для приложений `create-react-app` в файл public/index.html следует вставить код, пример которого показан в листинге 6.7.

Листинг 6.7. Подключение ресурсов CSS библиотеки Bootstrap

```
<link
rel="stylesheet"
href="https://maxcdn.bootstrapcdn.com/bootstrap/4.5.0/css/bootstrap.min.css"
integrity="sha384-9aIt2nRpC12Uk9gS9baDl411NQApFmC26EwAOH8WgZl5MYYxFfc+NcPb1dKG"
crossorigin="anonymous"
/>
```

Будет крайне желательным заменить это самой последней доступной стабильной версией Bootstrap. Управление версиями Bootstrap нужно будет выполнять вручную, поскольку она не обновляется автоматически при обновлении библиотек JavaScript.

Библиотека Bootstrap хорошо подходит для решения общих задач, но ее особенно сильная сторона — поддержка форм. Разметка форм требуемым образом может занять много времени и усилий. Библиотека Bootstrap выполняет всю тяжелую работу, позволяя разработчику концентрироваться на функциональности формы. Компонент `Form` библиотеки React Bootstrap содержит почти все необходимые средства для создания форм. Например, подкомпонент `Form.Control` создает ввод `input` по умолчанию, подкомпонент `Form.Label` создает метку `label`, а подкомпонент `Form.Group` ассоциирует эти два компонента и размечает их должным образом (листинг 6.8).

Листинг 6.8. Использование подкомпонентов компонента Form

```
<Form.Group controlId="startupName">
  <Form.Label>Startup name</Form.Label>
  <Form.Control placeholder="No names ending in ...ly, please"/>
</Form.Group>
```

Поля формы обычно отображаются в одной строке и занимают всю доступную ширину экрана. Разместить больше одного поля в строке можно с помощью подкомпонента `Form.Row`, как показано в листинге 6.9.

Листинг 6.9. Использование подкомпонента Form.Row

```
<Form.Row>
  <Form.Group as={Col} controlId="startupName">
    <Form.Label>Startup name</Form.Label>
    <Form.Control placeholder="No names ending in ...ly, please"/>
  </Form.Group>
  <Form.Group as={Col} controlId="market">
    <Form.Label>Market</Form.Label>
    <Form.Control placeholder="e.g. seniors on Tik-Tok"/>
  </Form.Group>
</Form.Row>
```

Компонент `Col` обеспечивает правильные размеры меток и полей. Если нужно, чтобы поле формы применялось не для ввода, а для чего-либо другого, можно использовать атрибут `as`, как показано в листинге 6.10.

Листинг 6.10. Использование атрибута as

```
<Form.Control as="select" defaultValue="Choose...">
  <option>Progressive web application</option>
  <option>Conservative web application</option>
```

```
    <option>Android native</option>
    <option>iOS native</option>
    <option>New Jersey native</option>
    <option>VT220</option>
</Form.Control>
```

Код, приведенный в листинге 6.10, создает элемент `select` в стиле Bootstrap.

Сложенные все вместе (листинг 6.11), эти фрагменты кода создают форму, показанную на рис. 6.2.

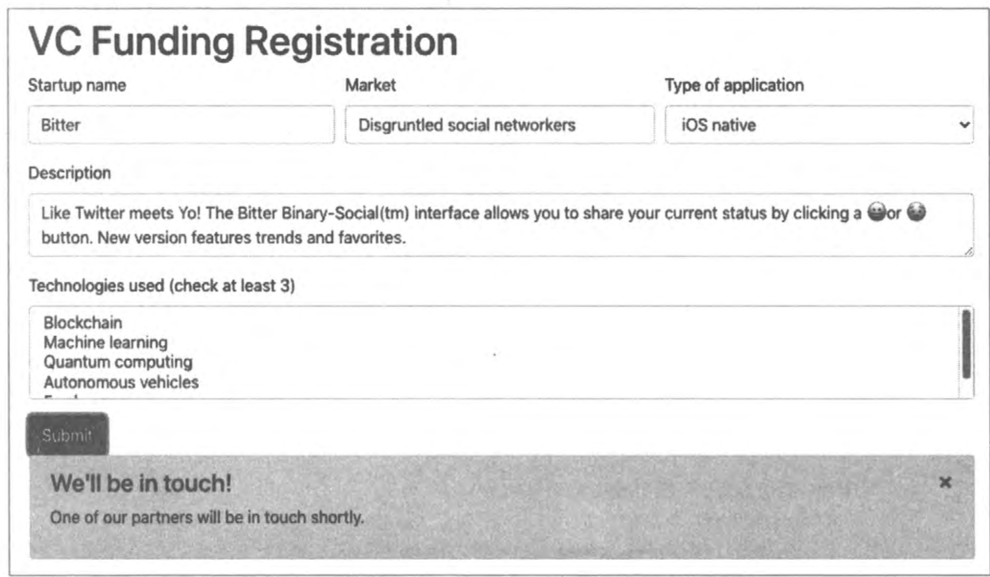


Рис. 6.2. Форма и диалоговое окно **Alert**, созданные посредством React Bootstrap

Листинг 6.11. Полный код формы, собранный из предыдущих фрагментов

```
import Form from 'react-bootstrap/Form'
import Col from 'react-bootstrap/Col'
import Button from 'react-bootstrap/Button'
import Alert from 'react-bootstrap/Alert'
import { useState } from 'react'
import './App.css'
function App() {
   const [submitted, setSubmitted] = useState(false)
   return (
      <div className="App">
         <h1>VC Funding Registration</h1>
         <Form>
            <Form.Row>
               <Form.Group as={Col} controlId="startupName">
                  <Form.Label>Startup name</Form.Label>
```

```
            <Form.Control placeholder=
             "No names ending in ...ly, please" />
         </Form.Group>
         <Form.Group as={Col} controlId="market">
            <Form.Label>Market</Form.Label>
            <Form.Control placeholder=
             "e.g. seniors on Tik-Tok" />
         </Form.Group>
         <Form.Group as={Col} controlId="appType">
            <Form.Label>Type of application</Form.Label>
            <Form.Control as="select" defaultValue="Choose...">
               <option>Progressive web application</option>
               <option>Conservative web application</option>
               <option>Android native</option>
               <option>iOS native</option>
               <option>New Jersey native</option>
               <option>VT220</option>
            </Form.Control>
         </Form.Group>
      </Form.Row>
      <Form.Row>
         <Form.Group as={Col} controlId="description">
            <Form.Label>Description</Form.Label>
            <Form.Control as="textarea" />
         </Form.Group>
      </Form.Row>
      <Form.Group id="technologiesUsed">
         <Form.Label>
            Technologies used (check at least 3)
         </Form.Label>
         <Form.Control as="select" multiple>
            <option>Blockchain</option>
            <option>Machine learning</option>
            <option>Quantum computing</option>
            <option>Autonomous vehicles</option>
            <option>For-loops</option>
         </Form.Control>
      </Form.Group>
      <Button variant="primary" onClick={() =>
      setSubmitted(true)}>
         Submit
      </Button>
   </Form>
   <Alert
      show={submitted}
      variant="success"
      onClose={() => setSubmitted(false)}
      dismissible
   >
```

```
            <Alert.Heading>We'll be in touch!</Alert.Heading>
            <p>One of our partners will be in touch shortly.</p>
         </Alert>
      </div>
   )
}
export default App
```

Обсуждение

Набор инструментов Bootstrap для создания пользовательского интерфейса намного старше, чем платформа Material Design, но до сих пор есть рыночные ниши, в которых он предпочтительнее. Если разрабатываемое приложение должно выглядеть как традиционный веб-сайт, этого можно добиться с помощью библиотеки Bootstrap. Если же вам нужно что-то, имеющее вид межплатформенного приложения, следует рассмотреть возможность использования библиотеки Material-UI[3].

Исходный код данного рецепта можно загрузить на веб-сайте GitHub по адресу **https://oreil.ly/ZpzF3**.

6.3. Просмотр наборов данных посредством окна React Window

ЗАДАЧА

Некоторые приложения отображают кажущийся бесконечный поток данных. Например, в приложениях наподобие Twitter мы не хотим отображать все сообщения в ленте, поскольку это может занять несколько часов, дней или даже месяцев. Решение этой проблемы — отображать данные отдельными окнами. При отображении окна со списком элементов в памяти размещаются только отображаемые в настоящее время элементы. При прокрутке вверх или вниз приложение загружает данные, требующиеся для текущего представления.

Но создание этой оконной логики — довольно сложная задача. Нам нужно не только тщательно отслеживать данные, отображаемые в настоящее время[4], но также быть осторожным и эффективно кешировать оконные данные, чтобы не возникали проблемы с памятью.

Как же можно реализовать возможность оконного просмотра данных в приложении React?

РЕШЕНИЕ

Данную задачу можно решить посредством библиотеки React Window. Эта библиотека представляет собой набор компонентов для приложений, в которых нужно

---

[3] Дополнительная информация предоставляется в *разделе 5.1*.

[4] Включая обработку всех неприглядных граничных случаев, возникающих при изменении размера окна.

выполнять прокрутку больших массивов данных. Рассмотрим, как при помощи этой библиотеки создать большой список фиксированного размера[5].

Для начала нам нужно создать компонент, который будет отображать подробности одного элемента списка. В нашем примере приложения мы создадим набор из 10 000 строк дат. Каждая дата будет отрисовываться компонентом DataRow, который станет нашим отрисовщиком элементов. Библиотека React Window отрисовывает только те элементы, которые присутствуют в текущем окне просмотра. При прокрутке списка вверх или вниз она будет создавать новые элементы по мере их появления в окне и удалять те, которые выводятся из окна.

При вызове отрисовщика элементов библиотека React Window передает ему два свойства: номер элемента, начинающийся с цифры 0, и объект стиля. В листинге 6.12 приведен код отрисовщика элементов DateRow.

Листинг 6.12. Код отрисовщика элементов DateRow

```
import moment from 'moment'
   const DateRow = ({ index, style }) => (
   <div className={`aDate ${index % 2 && 'aDate-odd'}`} style={style}>
      {moment().add(index, 'd').format('dddd, MMMM Do YYYY')}
   </div>
)
export default DateRow
```

Данный компонент вычисляет дату, находящуюся через index дней в будущем. Но в более реалистичном приложении этот компонент, скорее всего, загружал бы элемент даты с сервера бэкенда.

Для генерирования самого списка используем компонент FixedSizeList. Наш список должен иметь фиксированную ширину и высоту. Библиотека React Window вычисляет количество видимых в окне элементов по высоте окна и высоте каждого элемента на основе значения атрибута itemSize. Например, если высота окна height равна 400, а высота элемента itemHeight равна 40, то список должен содержать только 10 или 11 компонентов DateRow (рис. 6.3).

В листинге 6.13 приведена конечная версия кода приложения.

Листинг 6.13. Конечная версия кода приложения

```
import { FixedSizeList } from 'react-window'
import DateRow from './DateRow'
import './App.css'
function App() {
```

[5] Посредством библиотеки можно создавать списки и матрицы как переменного, так и фиксированного размера. Дополнительная информация предоставляется в документации по библиотеке (**https://oreil.ly/pCYaq**).

```
  return (
    <div className="App">
      <FixedSizeList
        height={400}
        itemCount={10000}
        itemSize={40}
        width={300}
      >
        {DateRow}
      </FixedSizeList>
    </div>
  )
}
export default App
```

Рис. 6.3. Список содержит только видимые в окне элементы

Обратите внимание на то, что компонент `FixedSizeList` не содержит экземпляра компонента `DateRow`. Это объясняется тем, что для генерирования множественных элементов при прокрутке списка он должен использовать функцию `DateRow`. Поэтому вместо компонента `<DateRow/>` в списке применяется сама функция `{DateRow}`.

Осталось только указать, что поскольку элементы добавляются в список и удаляются из него динамически, необходимо соблюдать осторожность при использовании селектора *n*-го потомка в CSS:

```
.aDate:nth-child(even) { /* Это неправильно */
  background-color: #eee;
}
```

Вместо этого нужно динамически проверять, является ли текущий индекс элемента нечетным, выполняя для этого деление по модулю 2, как показано в следующем примере:

```
<div className={`aDate ${index % 2 && 'aDate-odd'}`} ...>
```

Обсуждение

Библиотека компонентов React Window предназначена для решения узкого круга задач, но может быть полезной в случаях, когда нужно отображать огромные наборы данных. Ответственным за загрузку и кеширование отображаемых в списке данных продолжает оставаться разработчик, но это сравнительно простая задача по сравнению с отображением их динамически в окне, реализуемым библиотекой React Window.

Исходный код данного рецепта можно загрузить на веб-сайте GitHub по адресу **https://oreil.ly/lXxv3**.

6.4. Создание реагирующих диалоговых окон посредством библиотеки Material-UI

ЗАДАЧА

При использовании библиотеки компонентов существует высокая вероятность, что в определенный момент будет отображено диалоговое окно. Диалоговые окна позволяют вставлять дополнительную информацию в пользовательский интерфейс, не вызывая у пользователя чувства перехода на другую страницу. Они хорошо подходят для создания контента или как быстрое средство отображения дополнительной информации об элементе.

Но диалоговые окна не очень хорошо работают на мобильных устройствах. Мобильные устройства имеют небольшой экран, и диалоговые окна часто занимают много места у края для отображения фоновой страницы.

Как можно создать реагирующее диалоговое окно, отображающееся в виде всплывающего окна на настольных устройствах, но в виде отдельной полноэкранной страницы на мобильных?

РЕШЕНИЕ

Библиотека Material-UI содержит компонент высшего порядка (higher-order component) `withMobileDialog`, способный определять исполнение приложения на мобильном устройстве и соответственно отображать диалоговые окна как полноэкранные окна (листинг 6.14).

Листинг 6.14. Использование компонента `withMobileDialog`

```
import { withMobileDialog } from '@material-ui/core'
...
const ResponsiveDialog = withMobileDialog()(
  ({ fullScreen }) => {
    // Return some component using the fullScreen (true/false) property
  }
)
```

Компонент withMobileDialog придает любому заключенному в него компоненту дополнительное булево свойство fullScreen, посредством которого поведение компонента Dialog можно изменять. Например, если передать свойство fullScreen компоненту Dialog, как показано в листинге 6.15, то диалоговое окно будет отображаться по-разному при исполнении на мобильных и настольных устройствах.

Листинг 6.15. Передача свойства fullScreen компоненту Dialog

```
import { withMobileDialog } from '@material-ui/core'
import Dialog from '@material-ui/core/Dialog'
import DialogTitle from '@material-ui/core/DialogTitle'
import Typography from '@material-ui/core/Typography'
import DialogContent from '@material-ui/core/DialogContent'
import DialogActions from '@material-ui/core/DialogActions'
import Button from '@material-ui/core/Button'
import CloseIcon from '@material-ui/icons/Close'
const ResponsiveDialog = withMobileDialog()(
   ({ onClose, open, title, fullScreen, children }) => {
      return (
         <Dialog open={open} fullScreen={fullScreen} onClose={onClose}>
            <DialogTitle>
               <Typography
                  component="h1"
                  variant="h5"
                  color="inherit"
                  noWrap
               >
                  {title}
               </Typography>
            </DialogTitle>
            <DialogContent>{children}</DialogContent>
            <DialogActions>
               <Button
                  variant="outlined"
                  startIcon={<CloseIcon />}
                  onClick={onClose}
               >
                  Close
               </Button>
            </DialogActions>
         </Dialog>
      )
   }
)
export default ResponsiveDialog
```

Предположим, мы модифицируем приложение, созданное нами в *разделе 5.1*. В исходном приложении щелчок по изображению в галерее открывает диалоговое окно, которое на мобильном устройстве будет выглядеть, как показано на рис. 6.4.

Рис. 6.4. По умолчанию при исполнении на мобильном устройстве между краями диалогового окна и краями экрана есть пустое пространство

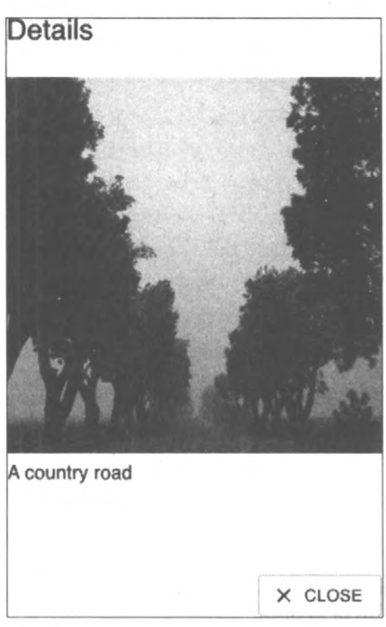


Рис. 6.5. Модифицированное диалоговое окно заполняет весь экран мобильного устройства

Если заменить стандартный компонент диалогового окна компонентом `ResponsiveDialog`, то на большом экране он будет отображаться таким же образом, но на мобильном устройстве заполнит весь экран, как показано на рис. 6.5.

Такой подход не только создает дополнительное пространство для отображения содержимого диалогового окна, но также упрощает его восприятие пользователем мобильного устройства. В частности, оно будет ощущаться как отдельная страница, а не как всплывающее окно.

Обсуждение

Дополнительные идеи по применению реагирующих интерфейсов рассматриваются в *разделе 2.1*.

Исходный код данного рецепта можно загрузить на веб-сайте GitHub по адресу **https://oreil.ly/836i2**.

6.5. Создание консоли администратора посредством React Admin

ЗАДАЧА

Разработчики уделяют так много времени разработке и поддержке приложений для конечных пользователей, что часто упускают из виду одну важную задачу: создание консоли администрирования приложения. Консоли администрирования предназначены не для клиентов, а для персонала бэк-офиса и администраторов, осуществляющих обзор текущих наборов данных и выясняющих причины проблем с данными в приложении для их устранения. Некоторые системы хранения данных, например Firebase (**https://oreil.ly/ugvEI**), оснащены довольно продвинутыми встроенными консолями администрирования. Но это далеко не стандартный подход в случае с большинством служб бэкенда. Вместо этого разработчикам нередко приходится выяснять причины проблем с данными, напрямую обращаясь к базам данных, которые часто находятся под несколькими уровнями облачной инфраструктуры.

Как же нам создать консоль администрирования почти для любого приложения React?

РЕШЕНИЕ

Эту задачу можно решить при помощи фреймворка React Admin. Хотя данная глава посвящена библиотекам компонентов, React Admin содержит намного больше, чем одни компоненты. Это фреймворк для разработки приложений, который облегчает задачу создания интерфейсов, предназначенных для исследования и обслуживания администраторами данных приложения.

Разные приложения используют разные уровни сетевых служб, например REST, GraphQL или еще какую-либо из многих других систем. Но в большинстве случаев доступ к данным реализуется как к набору ресурсов, хранящихся на сервере. Фреймворк React Admin содержит большинство компонентов, необходимых для создания приложения администрирования, которое позволит работать с каждым ресурсом, предоставляя возможности для создания и обслуживания данных, поиска требуемых данных, а также экспорта данных во внешние приложения.

Чтобы продемонстрировать работу фреймворка React Admin, мы создадим консоль для администрирования приложения доски объявлений (рис. 6.6), которое было разработано в *главе 5*.

Службы бэкенда для этого приложения предоставляются простым сервером GraphQL. Сервер GraphQL имеет сравнительно простую схему, которая определяет сообщения на языке описания схем, как показано в листинге 6.16.

Листинг 6.16. Описание сообщения на основе схемы

```
type Message {
  id: ID!
```

```
  author: String!
  text: String!
}
```

Каждому сообщению присваивается уникальный идентификатор `ID`, а в строковых переменных `text` и `author` сохраняются текст и автор сообщения соответственно.

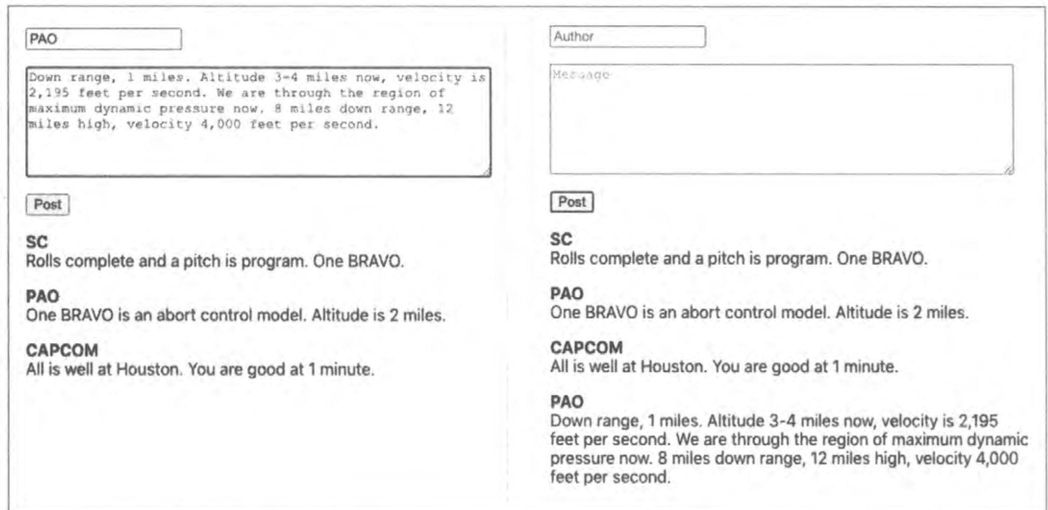


Рис. 6.6. Окно исходного приложения доски объявления

В исходном приложении пользователь мог изменять данные только одним способом, а именно добавляя сообщение. Также мог исполняться только один тип запросов: чтение всех сообщений.

Чтобы создать приложение с использованием React Admin, сначала нужно создать обычное приложение React, а затем в его папке установить библиотеку `react-admin`:

```
$ npm install react-admin
```

Основной компонент библиотеки называется `Admin`. Этот компонент будет оболочкой всего нашего приложения:

```
<Admin dataProvider={...}>
  ...Сюда вставляется пользовательский интерфейс для отдельных ресурсов...
</Admin>
```

Для работы компонента `Admin` требуется *источник данных* — адаптер, который будет подключать приложение к службе бэкенда. Наша служба бэкенда использует сервер GraphQL, поэтому нам нужен соответствующий источник данных:

```
$ npm install graphql
$ npm install ra-data-graphql-simple
```

Для большинства служб бэкенда существуют соответствующие источники данных. Подробная информация по этому вопросу предоставляется на веб-сайте фреймвор-

ка React Admin (**https://oreil.ly/2qtVY**). Прежде чем использовать источник данных, его нужно инициализировать. Конфигурирование сервера GraphQL осуществляется при помощи функции `buildGraphQLProvider` (листинг 6.17). Но поскольку эта функция работает в асинхронном режиме, то нужно следить за тем, чтобы она была в состоянии готовности, прежде чем вызывать ее.

Листинг 6.17. Конфигурирование сервера GraphQL

```
import { Admin } from 'react-admin'
import buildGraphQLProvider from 'ra-data-graphql-simple'
import { useEffect, useState } from 'react'
function App() {
   const [dataProvider, setDataProvider] = useState()
   useEffect(() => {
      let didCancel = false
      ;(async () => {
         const dp = await buildGraphQLProvider({
            clientOptions: { uri: 'http://localhost:5000' },
         })
         if (!didCancel) {
            setDataProvider(() => dp)
         }
      })()
      return () => {
         didCancel = true
      }
   }, [])

   return (
      <div className="App">
         {dataProvider && (
            <Admin dataProvider={dataProvider}>
               ...resource UI here...
            </Admin>
         )}
      </div>
   )
}
export default App
```

Источник данных подключается к серверу GraphQL, работающему на порту 5000[6]. После подключения источник данных сначала загружает схему для приложения,

[6] Код сервера можно найти в исходном коде для этой главы. Сервер запускается на исполнение командой `node ./server.js`.

которая информирует его о доступных ресурсах (в данном случае только один ресурс, `Messages`) и об операциях, которые он может выполнять с ними.

Если запустить приложение на данном этапе, то оно не будет ничего делать. Это объясняется тем, что хотя оно и знает о присутствии на сервере ресурса `Messages`, ему неизвестно, что с ним нужно что-либо делать. Поэтому нам нужно добавить ресурс `Messages` в приложение.

Чтобы приложение отображало список всех сообщений на сервере, нам нужно создать простой компонент `ListMessages` (листинг 6.18). Для формирования его интерфейса потребуется несколько готовых компонентов библиотеки `react-admin`.

Листинг 6.18. Создание компонента ListMessages

```
const ListMessages = (props) => {
  return (
    <List {...props}>
      <Datagrid>
        <TextField source="id" />
        <TextField source="author" />
        <TextField source="text" />
      </Datagrid>
    </List>
  )
}
```

С помощью этого компонента создается таблица со столбцами для идентификатора `id` сообщения, его автора `author` и текста `text`. Теперь можно сообщить системе администрирования о новом компоненте, передав атрибут `Resource` компоненту `Admin`:

```
<Admin dataProvider={dataProvider}>
  <Resource name="Message" list={ListMessages}/>
</Admin>
```

Компонент `Admin` обнаружит новый атрибут `Resource`, обратится к серверу и считает с него сообщения, которые затем отрисует посредством компонента `ListMessages` (рис. 6.7).

Может показаться, что обновление экрана осуществляется каким-то волшебством, но это потому, что сервер должен следовать определенным соглашениям, чтобы адаптер GraphQL знал, какие службы вызывать. В данном случае он находит запрос `allMessages`, который и возвращает сообщения (листинг 6.19).

Листинг 6.19. Код запроса allMessages

```
type Query {
  Message(id: ID!): Message
  allMessages(page: Int, perPage: Int,
    sortField: String, sortOrder: String,
    filter: MessageFilter): [Message]
}
```

Messages

Messages EXPORT

Id ↑	Author	Text
3bc62804-1aff-4d9b-bad8-bfbb0602b6d5	CAPCOM	All is well at Houston. You are good at 1 minute.
2bc8d451-d843-450e-91d1-2799b60f9744	SC	Rolls complete and a pitch is program. One BRAVO.
2aaf5636-5db9-415a-90f3-581749689ab8	PAO	One BRAVO is an abort control model. Altitude is 2 miles.

Rows per page: 10 1-3 of 3

Рис. 6.7. Отображение сообщений, полученных с сервера

В результате вам может потребоваться изменить API-интерфейс бэкенда, чтобы он соответствовал требованиям вашего источника данных. Но добавленные службы, скорее всего, будут полезными для основного приложения.

Запрос `allMessages` позволяет интерфейсу администрирования обрабатывать постранично данные с сервера. Он может принимать свойство `filter` и использовать его для поиска данных. Фильтр `MessageFilter` в схеме примера приложения позволит консоли администрирования находить сообщения, содержащие строковые значения в переменных `author` и `text`. Он также даст возможность отправлять общую строку поиска (`q`) для поиска сообщений, содержащих строковое значение в любом поле.

В листинге 6.20 приведено определение объекта `MessageFilter`. Подобный объект фильтра нужно создавать для каждого ресурса разрабатываемого вами приложения.

Листинг 6.20. Определение объекта фильтра `MessageFilter`

```
input MessageFilter {
  q: String
  author: String
  text: String
}
```

Чтобы разрешить фильтрацию и поиск на фронтенде, нужно сначала создать несколько полей фильтрации в компоненте, который мы назовем `MessageFilter` (листинг 6.21). Несмотря на одинаковое название, это совсем иная сущность, чем одноименный объект в схеме, хотя она и содержит поля с одинаковыми названиями.

Листинг 6.21. Определение компонента `MessageFilter`

```
const MessageFilter = (props) => (
  <Filter {...props}>
    <TextInput label="Author" source="author" />
```

```
      <TextInput label="Text" source="text" />
      <TextInput label="Search" source="q" alwaysOn />
   </Filter>
)
```

Теперь мы можем добавить компонент `MessageFilter` в компонент `ListMessages`, в результате чего сможем постранично просматривать сообщения, фильтровать их и искать требуемые (рис. 6.8).

Листинг 6.22. Добавление компонента `MessageFilter` в компонент `ListMessages`

```
const ListMessages = (props) => {
   return (
      <List {...props} filters={<MessageFilter />}>
         <Datagrid>
            <TextField source="id" />
            <TextField source="author" />
            <TextField source="text" />
         </Datagrid>
   </List>
   )
}
```

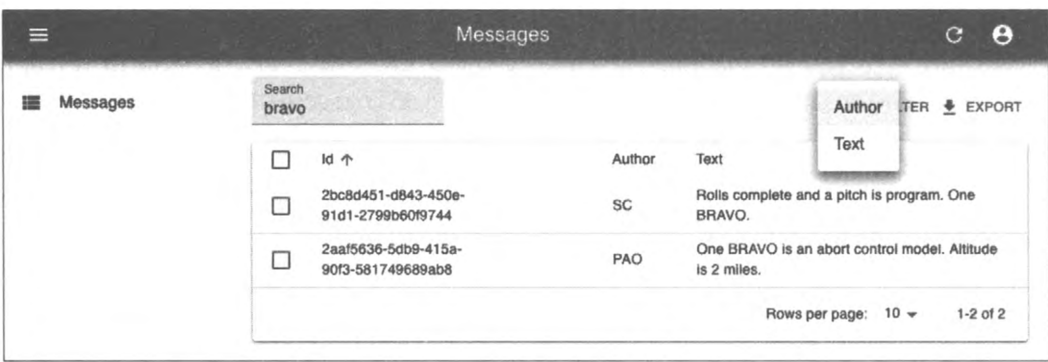


Рис. 6.8. Фильтрация сообщений в таблице по автору или тексту

Также можно получить возможность создавать новые сообщения. Для этого сначала создадим компонент `CreateMessage`, как показано в листинге 6.23.

Листинг 6.23. Создание компонента `CreateMessage`

```
const CreateMessage = (props) => {
   return (
      <Create title="Create a Message" {...props}>
         <SimpleForm>
            <TextInput source="author" />
```

```
            <TextInput multiline source="text" />
         </SimpleForm>
      </Create>
   )
}
```

Затем добавим этот компонент в атрибут `Resource`:

```
<Resource name="Message" list={ListMessages} create={CreateMessage}/>
```

На рис. 6.9 показан конечный результат.

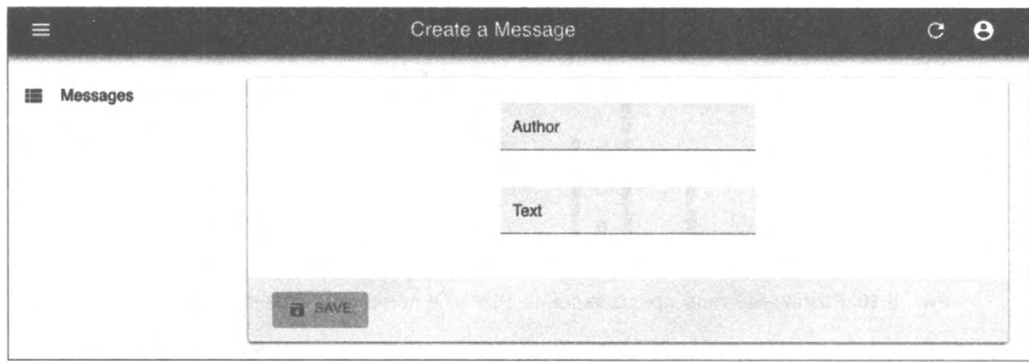


Рис. 6.9. Создание сообщений в консоли администрирования

Сообщения будет создавать источник данных GraphQL, передавая содержимое формы `CreateMessage` мутации, называющейся `CreateMessage` (листинг 6.24).

Листинг 6.24. Мутация `CreateMessage`

```
type Mutation {
   createMessage(
      author: String!
      text: String!
   ): Message
}
```

Подобным образом можно добавить возможность обновлять или удалять сообщения. При наличии сложной схемы с подресурсами средствами `react-admin` можно отображать подэлементы в таблице, а также обрабатывать дисплеи разных типов и отображать изображения и ссылки. Фреймворк React Admin также содержит компоненты для отображения ресурсов на календарях или в графиках. На рис. 6.10 показан пример такого отображения онлайновым демонстрационным приложением (**https://oreil.ly/fmFwR**)[7]. Консоли администрирования также могут работать совместно с установленными системами безопасности.

[7] Некоторые из таких компонентов доступны только подписчикам на версию фреймворка для предприятий.

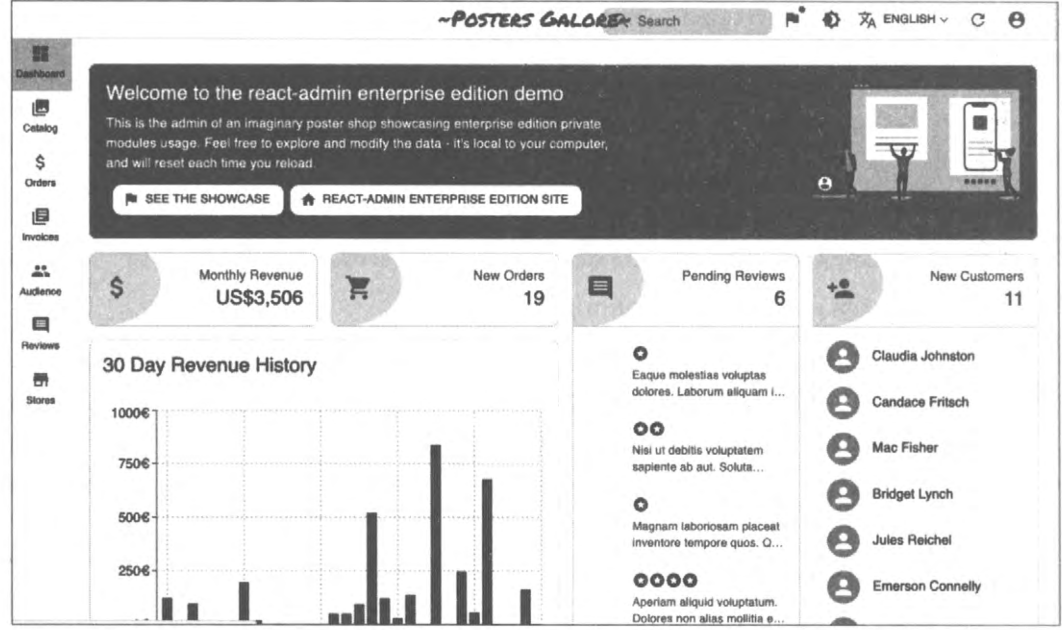


Рис. 6.10. Различные типы представлений данных в демонстрационном приложении

Обсуждение

Хотя для работы консоли администрирования на основе `react-admin` необходимо модифицировать бэкенд, добавив дополнительные службы, существует высокая вероятность, что эти новые службы также пригодятся и для основного приложения. Но даже если и не пригодятся, то предоставляемые этим фреймворком компоновочные блоки, скорее всего, сократят время, требуемое для разработки системы бэк-офиса.

Исходный код данного рецепта можно загрузить на веб-сайте GitHub по адресу **https://oreil.ly/2sUhp**.

6.6. Использование Semantic UI вместо дизайнера

ЗАДАЧА

Качественное стилистическое оформление может значительно повысить визуальную привлекательность приложения. И наоборот, плохое стилистическое оформление способно заставить даже хорошее приложение выглядеть дешевой любительской разработкой. У многих разработчиков[8] очень ограниченные дизайнерские

[8] В число которых входит, по крайней мере, один из авторов этой книги...

способности. Для таких разработчиков, которые не могут позволить себе привлечь профессионального дизайнера, наличие простой и четкой библиотеки компонентов пользовательского интерфейса позволит концентрироваться на функциональности приложения, не тратя бесконечные часы на корректирование расположения кнопок и границ.

Проверенные временем фреймворки типа Bootstrap могут предоставить хотя и простую, но качественную основу для большинства приложений[9]. Но даже такие фреймворки требуют уделять много внимания внешнему виду. Для тех, кто хочет сконцентрироваться на функциональности приложения, но при этом получить аккуратный практический внешний вид, хорошим решением будет использование библиотеки Semantic UI.

Но библиотека Semantic UI уже довольно старая, берет свое начало во времена царствования фреймворка jQuery. На момент работы над материалом данной книги она не обновлялась свыше двух лет. Так что же нам делать, если мы хотим совместить с React эту надежную, проверенную временем библиотеку Semantic US?

РЕШЕНИЕ

Данная задача решается посредством использования библиотеки Semantic UI React. Эта библиотека представляет собой оболочку, которая делает библиотеку Semantic UI доступной для приложений React.

Как можно судить по ее названию, библиотека Semantic UI концентрируется на значении интерфейса. В ней управление внешним видом осуществляется средствами CSS, а не через компоненты. А компоненты библиотеки Semantic UI, наоборот, концентрируются на функциональности. Например, при создании формы мы просто указываем, какие поля она должна содержать, не говоря ничего о том, где на форме их размещать. Это позволяет получить опрятный и единообразный внешний вид, требующий очень незначительной ручной корректировки, или вообще без нее.

Чтобы начать работать с библиотекой Semantic UI, нужно установить саму библиотеку и составляющую для поддержки стилистического оформления:

```
$ npm install semantic-ui-react semantic-ui-css
```

Также необходимо вставить ссылку на таблицу стилей в файл index.js приложения, как показано в листинге 6.25.

Листинг 6.25. Подключение таблицы стилей

```
import React from 'react'
import ReactDOM from 'react-dom'
import './index.css'
import App from './App'
```

[9] Руководство по использованию библиотеки Bootstrap в разрабатываемом приложении приводится в *разделе 6.2*.

```
import reportWebVitals from './reportWebVitals'
import 'semantic-ui-css/semantic.min.css'
ReactDOM.render(
   <React.StrictMode>
      <App />
   </React.StrictMode>,
   document.getElementById('root')
)
// Если вы хотите выполнять замеры поддерживаемых метрик производительности
// в своем приложении, передайте соответствующую функцию в функцию
// reportWebVitals(), чтобы сохранить результаты в журнале (например,
// reportWebVitals(console.log)), или же отправьте ее
// на узел аналитики(Google). Дополнительную информацию
// см. по адресу: https://bit.ly/CRA-vitals
reportWebVitals()
```

Для демонстрации работы с библиотекой Semantic UI React мы воссоздадим наше старое приложение для отправки сообщений. Нам потребуется форма с текстовыми полями для ввода имени автора сообщения и для ввода самого сообщения. Компоненты библиотеки Semantic UI разработаны так, чтобы быть как можно более похожими на простые элементы HTML. Таким образом, при создании формы мы импортируем компоненты `Form`, `Input`, `TextArea` и `Button` для отправки сообщений (листинг 6.26).

Листинг 6.26. Форма для ввода сообщений

```
import { Button, Form, Input, TextArea } from 'semantic-ui-react'
import './App.css'
import { useState } from 'react'
function App() {
   const [author, setAuthor] = useState('')
   const [text, setText] = useState('')
   const [messages, setMessages] = useState([])
   return (
      <div className="App">
         <Form>
            <Form.Field>
               <label htmlFor="author">Author</label>
               <Input
                  value={author}
                  id="author"
                  onChange={(evt) => setAuthor(evt.target.value)}
               />
            </Form.Field>
            <Form.Field>
               <label htmlFor="text">Message</label>
```

```
            <TextArea
              value={text}
              id="text"
              onChange={(evt) => setText(evt.target.value)}
            />
          </Form.Field>
          <Button
            basic
            onClick={() => {
              setMessages((m) => [
                {
                  icon: 'pencil',
                  date: new Date().toString(),
                  summary: author,
                  extraText: text,
                },
                ...m,
              ])
              setAuthor('')
              setText('')
            }}
          >
            Post
          </Button>
        </Form>
      </div>
  )
}
export default App
```

Этот код должен быть вам знаком. Компонент `Form` имеет вспомогательное поле `Field`, что немного упрощает группировку меток и полей, но, помимо этого, этот код выглядит похожим на код обычной формы HTML.

В примере приложения мы "отправляем" сообщения, добавляя их в массив `messages`. Возможно, вы заметили, что сообщения добавляются в этот массив в определенную объектную структуру, показанную в листинге 6.27.

Листинг 6.27. Структура для добавления сообщений

```
setMessages((m) => [
  {
    icon: 'pencil',
    date: new Date().toString(),
    summary: author,
    extraText: text,
  },
  ...m,
])
```

Эти атрибуты не были выбраны каким-либо случайным образом. Хотя большинство компонентов библиотеки Semantic UI простого типа, она также содержит некоторые более сложные компоненты, предназначенные для поддержки ряда распространенных ситуаций. Компонент `Feed` — один из таких. Он предназначен для отрисовки потока сообщений, наподобие сообщений в социальной сети Twitter или Instagram. Компонент отрисовывает последовательность сообщений, включая метки времени, заголовки, значки и т. д. Конечный код нашего приложения с включением в него компонента `Feed` приведен в листинге 6.28.

Листинг 6.28. Конечный код приложения

```
import {
   Button,
   Form,
   Input,
   TextArea,
   Feed,
} from 'semantic-ui-react'
import './App.css'
import { useState } from 'react'
function App() {
   const [author, setAuthor] = useState('')
   const [text, setText] = useState('')
   const [messages, setMessages] = useState([])
   return (
   <div className="App">
   <Form>
      <Form.Field>
         <label htmlFor="author">Author</label>
         <Input
            value={author}
            id="author"
            onChange={(evt) => setAuthor(evt.target.value)}
         />
      </Form.Field>
      <Form.Field>
         <label htmlFor="text">Message</label>
         <TextArea
            value={text}
            id="text"
            onChange={(evt) => setText(evt.target.value)}
         />
      </Form.Field>
      <Button
         basic
```

```
        onClick={() => {
            setMessages((m) => [
                {
                    icon: 'pencil',
                    date: new Date().toString(),
                    summary: author,
                    extraText: text,
                },
                ...m,
            ])
            setAuthor('')
            setText('')
        }}
    >
        Post
    </Button>
    </Form>
    <Feed events={messages} />
  </div>
 )
}
export default App
```

В результате исполнения этого кода получаем простой и аккуратно оформленный интерфейс (рис. 6.11).

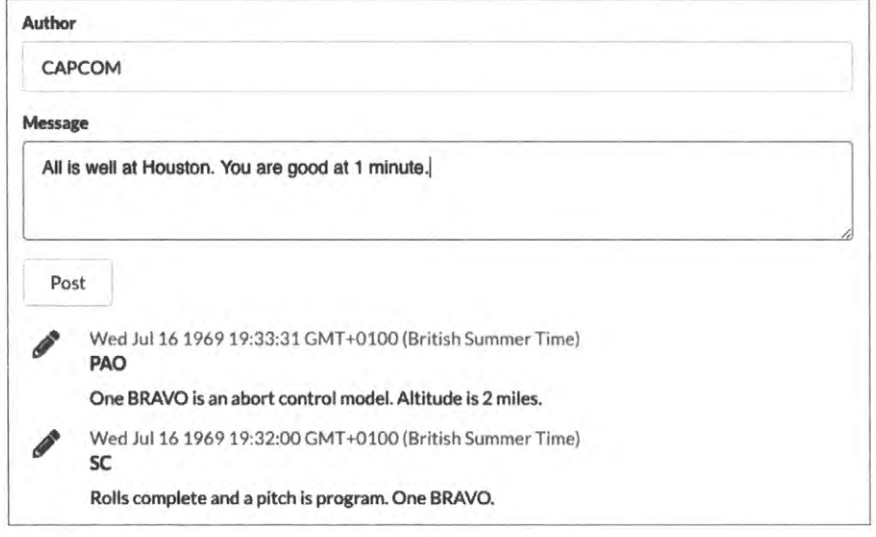


Рис. 6.11. Интерфейс, созданный при помощи библиотеки Semantic UI

Обсуждение

Хотя библиотека Semantic UI довольно старая, это не означает, что она плохая. Ее проверенные временем средства позволяют создавать приятный и функциональный интерфейс. Это один из наилучших способов создать полноценное приложение, не прибегая к услугам визуального дизайнера. Эта библиотека особенно полезна при создании приложений типа Lean Startup, когда нужно быстро разработать тестовое приложение, чтобы проверить, существует ли рынок для вашего продукта[10].

Исходный код данного рецепта можно загрузить на веб-сайте GitHub по адресу **https://oreil.ly/qeNqy**.

[10] Дополнительная информация на эту тему приводится в книге *The Lean Startup*, автор Эрик Райс (Eric Ries), издательство Crown Business.

ГЛАВА 7

Безопасность

В этой главе мы рассмотрим разные способы придания безопасности разрабатываемым приложениям. Для этого мы опишем распространенные методы интегрирования приложения со стандартными системами обеспечения безопасности. Также мы выясним, как проверить код на отсутствие распространенных уязвимостей в системе безопасности. Во многих рецептах в этой главе мы будем использовать API-интерфейс стандарта WebAuthn для интегрирования приложения с устройствами безопасности, такими как сканер отпечатков пальцев и физические ключи. Эта интересная, но недостаточно применяемая технология может повысить как уровень безопасности разрабатываемого приложения, так и уровень восприятия приложения пользователем.

7.1. Защищаем запросы, а не маршруты

ЗАДАЧА

В *разделе 2.6* мы увидели, как можно на основе средств маршрутизации React создать безопасные маршруты. В частности, мы рассмотрели, как предотвратить несанкционированный доступ к определенным путям приложения, заставляя пользователей заполнить форму входа в систему.

Защита маршрутов — это хороший, довольно общий подход на начальных этапах разработки приложения. Но некоторые приложения не поддаются легкому включению в эту статическую модель безопасности. Некоторые страницы будут защищенными, а некоторые — нет. Но во многих приложениях легче защитить службы данных, а не страницы. Иными словами, важно не то, на какой странице находится пользователь, а то, какие данные он просматривает.

Все эти задачи обычно можно прямолинейно решить на уровне API-интерфейса. Но механизмы защиты нежелательно воспроизводить в алгоритмах клиента фронтенда. По этим причинам простой подход с обозначением одних страниц как требующих защиты, а других — нет, не будет достаточно хорошим.

РЕШЕНИЕ

Если классификации маршрутов на требующие и не требующие защиты недостаточно для обеспечения безопасности разрабатываемого клиента, то полезно рассмотреть возможность управления доступом к приложению, используя запросы на подтверждения, получаемые от сервера бэкенда.

При таком подходе изначально предполагается, что пользователю доступны все маршруты приложения, Мы не разделяем маршруты на требующие и не требующие защиты, а просто работаем с маршрутами. Если пользователь посещает маршрут, содержащий закрытые данные, сервер API возвращает ошибку, обычно статус HTTP 401 (Доступ запрещен). Когда происходит такая ошибка, система безопасности перенаправляет пользователя на форму входа на защищенную страницу.

При этом подходе сервер API устанавливает политику, какие маршруты являются закрытыми, а какие — открытыми. В случае изменения политик безопасности нужно только откорректировать должным образом код на сервере API, не меняя код клиента.

Давайте освежим в памяти исходный рецепт защищенных маршрутов. В нашем приложении мы вставляем в файл App.js компонент `SecurityProvider`, который управляет безопасностью всех своих дочерних компонентов (листинг 7.1)

Листинг 7.1. Добавление компонента `SecurityProvider` в файл App.js

```
import './App.css'
import { BrowserRouter, Route, Switch } from 'react-router-dom'
import Public from './Public'
import Private1 from './Private1'
import Private2 from './Private2'
import Home from './Home'
import SecurityProvider from './SecurityProvider'
import SecureRoute from './SecureRoute'
function App() {
  return (
    <div className="App">
      <BrowserRouter>
        <SecurityProvider>
          <Switch>
            <Route exact path="/">
              <Home />
            </Route>
            <SecureRoute path="/private1">
              <Private1 />
            </SecureRoute>
            <SecureRoute path="/private2">
              <Private2 />
            </SecureRoute>
            <Route exact path="/public">
              <Public />
            </Route>
          </Switch>
        </SecurityProvider>
      </BrowserRouter>
```

```
        </div>
    )
}
export default App
```

Можно видеть, что приложение содержит открытые маршруты `Routes` и закрытые маршруты `SecuredRoutes`. При попытке пользователя получить доступ к закрытому маршруту он перенаправляется на форму для входа на данную страницу (рис. 7.1).

Пользователь может получить доступ к закрытой странице (рис. 7.2), предоставив на этой форме правильные имя пользователя и пароль

Login Page

You need to log in. (hint: try fred/password)

Username:
Password:
Login

Рис. 7.1. При попытке получить доступ к закрытому маршруту необходимо заполнить форму входа

Рис. 7.2. Предоставив правильные имя и пароль, пользователь получает доступ к закрытой странице

Для защиты, предоставляемой API-интерфейсом бэкенда, все закрытые маршруты `SecuredRoutes` нужно заменить открытыми страницами `Routes`. Приложение просто не будет знать, какие данные являются закрытыми, а какие — открытыми до тех пор, пока сервер API-интерфейса не предоставит ему эту информацию. Приложение в этом рецепте будет иметь две страницы, содержащие смесь открытых и закрытых данных. Страница *Transactions* будет считывать с сервера закрытые данные, а страница *Offers* — открытые. Модифицированный соответствующим образом файл App.js будет выглядеть, как показано в листинге 7.2.

Листинг 7.2. Модифицированный файл App.js

```
import './App.css'
import { BrowserRouter, Route, Switch } from 'react-router-dom'
import Transactions from './Transactions'
import Offers from './Offers'
import Home from './Home'
import SecurityProvider from './SecurityProvider'
function App() {
    return (
        <div className="App">
            <BrowserRouter>
                <SecurityProvider>
                    <Switch>
                        <Route exact path="/">
```

```
                <Home />
            </Route>
            <Route exact path="/transactions">
                <Transactions />
            </Route>
            <Route exact path="/offers">
                <Offers />
            </Route>
          </Switch>
        </SecurityProvider>
      </BrowserRouter>
    </div>
  )
}
export default App
```

Нам также нужно внести изменения в компонент `SecurityProvider`. В модели безопасности на основе API-интерфейса клиент начинает работу, предполагая, что все данные находятся в открытом доступе, что противоположно подходу с закрытыми маршрутами, в котором предполагается отсутствие права доступа, пока пользователь не докажет обратное, предоставив правильные учетные данные.

Это означает, что в исходном состоянии входа на страницу наш новый компонент `SecurityProvider` должен по умолчанию иметь значение `true` (листинг 7.3).

Листинг 7.3. Модифицированный компонент `SecurityProvider`

```
import { useState } from 'react'
import SecurityContext from './SecurityContext'
import Login from './Login'
import axios from 'axios'
const SecurityProvider = (props) => {
  const [loggedIn, setLoggedIn] = useState(true)
  return (
    <SecurityContext.Provider
      value={{
        login: async (username, password) => {
          await axios.post('/api/login', { username, password })
          setLoggedIn(true)
        },
        logout: async () => {
          await axios.post('/api/logout')
          return setLoggedIn(false)
        },
        onFailure() {
          return setLoggedIn(false)
        },
```

```
            loggedIn,
         }}
      >
         {loggedIn ? props.children : <Login />}
      </SecurityContext.Provider>
   )
}
export default SecurityProvider
```

В исходное приложение также были внесены следующие дополнительные изменения:

- Код, принимающий решение, должен ли пользователь заполнять форму входа, был перемещен в компонент `SecurityProvider`. В исходном приложении этот код находился в компоненте `SecuredRoute`, но теперь он исполняется централизованно.
- Мы заменили имитаторы проверки имени пользователя и пароля вызовами служб бэкенда `/api/login` и `/api/logout`. Но лучше всего будет заменить их кодом для предоставления доступа, используемым в вашей системе.
- Компонент `SecurityProvider` теперь предоставляет новую функцию `onFailure`, которая просто помечает пользователя как вышедшего из закрытой страницы.

При вызове этой функции она заставляет пользователя выполнить вход. Но если у нас больше нет закрытых маршрутов, то в какой точке мы выполняем проверку для предоставления доступа? Теперь это осуществляется в самих вызовах API-интерфейса.

В реальном приложении рекомендуется добавить код для обработки неуспешных попыток входа. Здесь мы опустили такой фрагмент, чтобы уменьшить объем кода приложения. При неуспешной попытке входа пользователь остается на странице формы входа без отображения каких-либо сообщений об ошибке.

Теперь посмотрим, как выглядит код нового компонента `Transactions`, хранящегося в файле src/Transactions.js (листинг 7.4). Этот компонент считывает данные транзакций и отображает их на экране.

Листинг 7.4. Код компонента `Transactions`

```
import useTransactions from './useTransactions'
const Transactions = () => {
   const { data: transactions } = useTransactions()
   return (
      <div>
         <h1>Transactions</h1>
         <main>
            <table>
               <thead>
                  <tr>
                     <th>Date</th>
```

```
                <th>Amount</th>
                <th>Description</th>
             </tr>
          </thead>
          <tbody>
             {transactions &&
                transactions.map((trx) => (
                   <tr>
                      <td>{trx.date}</td>
                      <td>{trx.amount}</td>
                      <td>{trx.description}</td>
                   </tr>
                ))}
             </tbody>
          </table>
       </main>
    </div>
  )
}
export default Transactions
```

Хук useTransactions содержит сетевой код для считывания данных с сервера. Как раз в этот хук нам нужно добавить проверку на ответ 401 (Запрещено) от сервера (листинг 7.5).

Листинг 7.5. Добавление в хук useTransaction проверки на ответ от сервера

```
import { useEffect, useState } from 'react'
import axios from 'axios'
import useSecurity from './useSecurity'

const useTransactions = () => {
   const security = useSecurity()
   const [transactions, setTransactions] = useState([])
   useEffect(() => {
      ;(async () => {
         try {
            const result = await axios.get('/api/transactions')
            setTransactions(result.data)
         } catch (err) {
            const status = err.response && err.response.status
            if (status === 401) {
               security.onFailure()
            }
            // Сюда вставляется код для обработки других исключений. (Рассмотрите
               возможность использования общего обработчика ошибок. Подробности
               см. в другом месте в этой книге.)
         }
```

```
    })()
  }, [])
  return { data: transactions }
}
export default useTransactions
```

В приложении для обращения к серверу мы используем библиотеку `axios`. Эта библиотека обрабатывает как исключения такие ошибки HTTP, как ошибка 401 (статус HTTP Unauthorized — Запрещено). В результате немного легче определить, какой код обрабатывает неожиданный ответ. При использовании API-интерфейса другого стандарта, например GraphQL, обработку ошибок безопасности можно реализовать аналогичным образом, исследуя содержимое объекта ошибок, возвращаемого сервером GraphQL.

При получении от сервера ответа, что доступ к ресурсу запрещен, хук `useTransactions` вызывает функцию `onFailure` из компонента `SecurityProvider`.

Страница **Offers** создается таким образом. Код в файле src/Offers.js будет заниматься форматированием данных `offers`, предоставляемых сервером (листинг 7.6).

Листинг 7.6. Код компонента `Offers`

```
import useOffers from './useOffers'
const Offers = () => {
  const { data: offers } = useOffers()
  return (
    <div>
      <h1>Offers</h1>
      <main>
        <ul>
          {offers &&
            offers.map((offer) => <li className="offer">{offer}</li>)}
        </ul>
      </main>
    </div>
  )
}
export default Offers
```

А данные считываются хуком `useOffers`, код которого находится в файле src/useOffers.js (листинг 7.7).

Листинг 7.7. Код хука `useOffers`

```
import { useEffect, useState } from 'react'
import axios from 'axios'
import useSecurity from './useSecurity'
```

```
const useOffers = () => {
  const security = useSecurity()
  const [offers, setOffers] = useState([])
  useEffect(() => {
    ;(async () => {
    try {
      const result = await axios.get('/api/offers')
      setOffers(result.data)
      } catch (err) {
        const status = err.response && err.response.status
        if (status === 401) {
          security.onFailure()
        }
        // Сюда вставляется код для обработки других исключений. (Рассмотрите
          возможность использования общего обработчика ошибок. Подробности
          см. в другом месте в этой книге.)
      }
    })()
  }, [])
  return { data: offers }
}
export default useOffers
```

Хотя конечная точка /api/offers не защищена, у нас предусмотрен код, который проверяет на наличие нарушений безопасности. Одно из следствий подхода к обеспечению безопасности с использованием API-интерфейса — все конечные точки должны рассматриваться как защищенные просто на случай, если они станут защищенными в будущем.

При запуске нашего приложения открывается домашняя страница (рис. 7.3).

Рис. 7.3. Домашняя страница приложения

Offers

- Free lard
- Buy one, get one same price
- Buy two, get one

Рис. 7.4. Список предложений, отображаемый при щелчке по ссылке **Offers**

При щелчке по ссылке **Offers** отображается список предложений, считанный с сервера (рис. 7.4). Эти данные открытые, поэтому нам не нужно выполнять вход для их просмотра.

Если теперь возвратиться обратно на домашнюю страницу и щелкнуть по ссылке **Transactions**, то приложение предложит нам выполнить вход на эту страницу, открыв соответствующее окно (рис. 7.5).

При попытке загрузки данных страницы транзакций с сервера появляется ответ 401 (Запрещено). Программа рассматривает это как исключение и вызывает функцию `onFailure` из компонента `SecurityProvide`, которая и отображает форму входа на рис. 7.5.

Login Page

You need to log in. (hint: try fred/password)

Username:
Password:
Login

Рис. 7.5. Для доступа к странице **Transactions** необходимо предоставить правильные учетные данные

Transactions

Date	Amount	Description
2023-12-04	3.45	Coffee
2023-12-05	6.15	Beard oil

Рис. 7.6. После успешного входа страница **Transactions** становится доступной

После ввода имени пользователя и пароля приложение отправляет их на сервер. При условии, что это не вызывает ошибку, компонент `SecurityProvider` скрывает форму входа, выполняется повторная отрисовка страницы **Transactions** (рис. 7.6), и мы получаем доступ к ее данным, поскольку выполнили вход.

Обсуждение

Теперь наше приложение не содержит ничего, что указывало бы, какие API-интерфейсы закрытые, а какие — нет. Вся эта работа выполняется на сервере. Безопасность приложения теперь полностью в ведении конечных точек API-интерфейса.

При таком подходе все вызовы API должны обрабатываться единообразно в аспекте безопасности. Одно из достоинств размещения вызовов API в специализированных хуках состоит в возможности использования всеми хуками общего кода обеспечения безопасности. Хуки могут вызывать другие хуки, при этом общий подход заключается в создании хуков, работающих как вызовы общего назначения `GET` и `POST`[1]. Хук общего назначения `GET` может не только обрабатывать ошибки доступа, но также содержать запросы отмены, очистку сетевых запросов (см. *разделы 5.3* и *5.6*), а также общую обработку ошибок (см. *раздел 4.1*).

Еще одно достоинство подхода с использованием API-интерфейса для обеспечения безопасности — возможность полного отключения защиты в некоторых обстоятельствах. Например, защиту можно отключить на стадии разработки приложения, устранив необходимость настройки поставщика идентификации для разработчиков. Также можно создавать различные конфигурации защиты для разных развертываний.

Наконец, для автоматизированных систем тестирования, например Cypress, которые могут эмулировать сетевые ответы, тестирование функциональности приложения можно отделить от тестирования нефункциональных аспектов, таких как обес-

[1] Или в случае GraphQL аксессоры и мутаторы.

печение безопасности. Целесообразно выполнить дополнительное тестирование безопасности только серверной стороны, отдельное от тестирования пользовательского интерфейса, чтобы убедиться в безопасности самого сервера.

Исходный код данного рецепта можно загрузить на веб-сайте GitHub по адресу **https://oreil.ly/ByWVZ**.

7.2. Аутентификация посредством физических ключей

ЗАДАЧА

Для обеспечения безопасности одних учетных данных не всегда достаточно, поскольку их могут украсть или угадать. Поэтому некоторые пользователи могут отдавать предпочтение только приложениям, предоставляющим дополнительный уровень безопасности.

В настоящее время все больше систем предоставляют возможность *двухфакторной аутентификации*. Система двухфакторной аутентификации требует, чтобы для входа пользователь сначала предоставил свои учетные данные, а затем еще какую-либо дополнительную информацию. Такой дополнительной информацией может быть код, отправленный пользователю в SMS-сообщении, или одноразовый пароль, создаваемый приложением на его смартфоне. Или же, что еще более безопасно, вторым фактором может быть использование физического устройства-ключа, например YubiKey (**https://www.yubico.com**), подключаемого в нужное время к компьютеру и активируемого нажатием кнопки.

Подобные физические ключи применяют шифрование с открытым ключом, в котором генерируется открытый ключ для данного приложения, а сообщения зашифровываются с помощью секретного ключа. Приложение может отправить устройству произвольную проверочную строку, генерируя подпись посредством секретного ключа. Затем приложение может использовать открытый ключ, чтобы проверить правильность подписи строки.

Но как можно интегрировать такие возможности физических ключей в приложения React?

РЕШЕНИЕ

Широко поддерживаемый[2] стандарт W3C для аутентификации в Интернете Web Authentication (также называется WebAuthn) позволяет браузеру взаимодействовать с физическими устройствами наподобие ключей YubiKey.

Аутентификация в Интернете осуществляется в два этапа. Первый этап называется *аттестацией* (attestation). В процессе аттестации пользователь регистрирует в приложении устройство обеспечения безопасности. А на втором этапе, называю-

[2] За исключением браузера Internet Explorer.

щимся *утверждением* (assertion), пользователь может подтвердить свою личность, чтобы войти в систему.

Рассмотрим сначала аттестацию. На этом этапе пользователь регистрирует физическое устройство по своей учетной записи. Это означает, что на всем протяжении этого этапа пользователь должен находиться в системе под своей учетной записью.

Код для данного рецепта содержит эмуляцию сервера Node, который можно запустить из папки server кода приложения:

```
$ cd server
$ npm install
$ npm run start
```

Аттестация содержит три шага:

1. Сервер генерирует запрос на аттестацию, указывая разрешенный тип устройства.
2. Пользователь подключает устройство и активирует его, например, нажимая на нем кнопку.
3. Устройство генерирует ответ, содержащий открытый ключ, который отправляется на сервер, где он сохраняется под учетной записью пользователя.

Узнать, поддерживает ли конкретный браузер стандарт WebAuthn, можно, проверив наличие интерфейса `window.PublicKeyCredential`.

Запрос на аттестацию на сервере создается конечной точкой /startRegister. Поэтому начнем с вызова этой точки:

```
import axios from 'axios'
...
// Запрос начать регистрацию физического ключа для текущего пользователя
const response = await axios.post('/startRegister')
```

В листинге 7.8 приведен код запроса на аттестацию.

Листинг 7.8. Код запроса на аттестацию

```
{
   "rpName": "Physical Token Server",
   "rpID": "localhost",
   "userID": "1234",
   "userName": "freda",
   "excludeCredentials": [
      {"id": "existingKey1", "type": "public-key"}
   ],
   "authenticatorSelection": {
      "userVerification": "discouraged"
   },
   "extensions": {
      "credProps": true
   }
}
```

Названия некоторых атрибутов начинаются с букв `rp...`, что означает *relaying party*, т. е. передающая сторона. Передающая сторона — это приложение, которое сгенерировало запрос.

Переменной `rpName` присваивается произвольная текстовая строка, описывающая приложение. Идентификатору `rpId` следует присвоить значение в виде имени текущего домена. В данном случае это `localhost`, поскольку мы работаем с сервером для разработки. Строка `userID` — однозначный идентификатор пользователя, а строка `userName` — имя пользователя.

Атрибут `excludeCredentials` довольно интересный. Пользователи могут регистрировать по своим учетным записям по несколько устройств. Значение данного атрибута — это список уже зарегистрированных устройств, чтобы не регистрировать одно и то же устройство дважды. При попытке зарегистрировать одно и то же устройство более одного раза браузер немедленно создаст исключение, сообщая, что устройство уже зарегистрировано где-то в другом месте.

Атрибут `authenticationSelection` позволяет задавать разные действия, которые пользователь должен выполнить при активировании своего устройства. В данном случае атрибуту `userVerification` присвоено значение `false`, чтобы не допустить выполнения пользователем каких-либо дополнительных шагов (например, ввода ПИН) при активировании устройства. Следовательно, при запросе подключить свое устройство к компьютеру пользователь вставит его в разъем USB и нажмет на нем кнопку, без необходимости выполнять какие-либо другие действия.

Расширение `credProps` запрашивает устройство возвратить дополнительные свойства учетных данных, которые могут быть полезными для сервера.

После того как сервер сгенерирует запрос аттестации, нужно дать указание пользователю подключить свое устройство безопасности к компьютеру. Это делается следующей функцией браузера:

```
navigator.credentials.create()
```

Функция `create()` принимает в качестве параметра объект запроса аттестации. К сожалению, данные в этом объекте должны быть разных двоичных типов низкого уровня, например массивы байтов. Мы можем значительно упростить себе жизнь, установив с депозитория GitHub библиотеку `webauthn-json`, которая позволит подавать запрос в формате JSON:

```
$ npm install "@github/webauthn-json"
```

Затем мы сможем передавать содержимое запроса WebAuthn версии функции `create()` с GitHub, как показано в листинге 7.9.

Листинг 7.9. Передача запроса WebAuthn функции `create()`

```
import { create } from '@github/webauthn-json'
import axios from 'axios'
...
// Запрос начать регистрацию физического ключа для текущего пользователя
const response = await axios.post('/startRegister')
```

```
// Передаем конфигурацию WebAuthn функции create() библиотеки webauthn-json
const attestation = await create({ publicKey: response.data })
```

В этой точке браузер дает указание пользователю подключить свое устройство безопасности к компьютеру и активировать его (рис. 7.7).

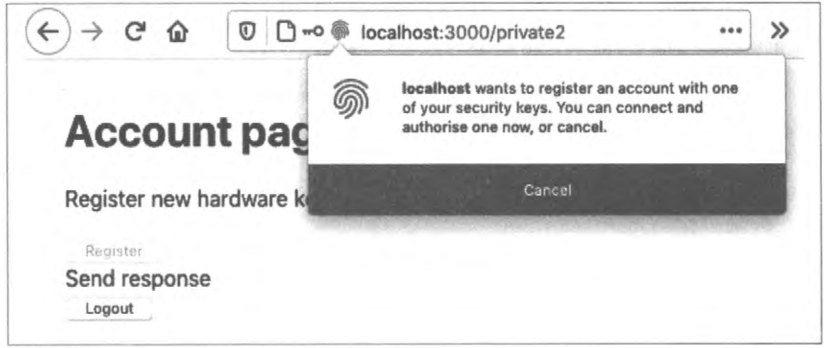


Рис. 7.7. При вызове функции `create()` браузер дает пользователю указание подключить физический ключ

Функция `create()` возвращает объект *аттестации*, который можно рассматривать, как регистрационную информацию для устройства. Сервер может использовать объект аттестации для проверки личности пользователя при его входе в систему. Нам нужно зарегистрировать данный объект аттестации по учетной записи пользователя. Для этого мы передаем его обратно концевой точке /register нашего сервера (листинг 7.10).

Листинг 7.10. Регистрация объекта аттестации

```
import { create } from '@github/webauthn-json'
import axios from 'axios'
...
// Запрос начать регистрацию физического ключа для текущего пользователя
const response = await axios.post('/startRegister')
// Передаем конфигурацию WebAuthn функции create() библиотеки webauthn-json
const attestation = await create({ publicKey: response.data })
// Отправляем подробности физического ключа YubiKey для регистрации по учетной записи
   пользователя
const attestationResponse = await axios.post('/register', {
   attestation,
})
```

Так выглядит процесс регистрации нового устройства безопасности для пользователя. Но куда его вставить в коде приложения?

Наше приложение имеет страницу *Account page* (рис. 7.8), и в эту страницу мы добавим кнопку для регистрации нового физического ключа.

А в листинге 7.11 приведен код для регистрации физического ключа.

Рис. 7.8. Кнопка для регистрации нового физического ключа на странице **Account page**

Листинг 7.11. Код для регистрации физического ключа

```
import { useState } from 'react'
import Logout from './Logout'
import axios from 'axios'
import { create } from '@github/webauthn-json'
const Private2 = () => {
   const [busy, setBusy] = useState(false)
   const [message, setMessage] = useState()
   return (
      <div className="Private2">
         <h1>Account page</h1>
         {window.PublicKeyCredential && (
            <>
               <p>Register new hardware key</p>
               <button
                  onClick={async () => {
                     setBusy(true)
                     try {
                        const response = await axios.post('/startRegister')
                        setMessage('Send response')
                        const attestation = await create({
                           publicKey: response.data,
                        })
                        setMessage('Create attestation')
                        const attestationResponse = await axios.post(
                           '/register',
                           {
                              attestation,
                           }
                        )
                        setMessage('registered!')
                        if (
                           attestationResponse.data &&
                           attestationResponse.data.verified
                        ) {
                           alert('New key registered')
                        }
```

```
                } catch (err) {
                    setMessage('' + err)
                } finally {
                    setBusy(false)
                }
            }}
            disabled={busy}
        >
            Register
        </button>
    </>
)}
<div className="Account-message">{message}</div>
<Logout />
</div>
)
}
export default Private2
```

При нажатии кнопки **Register** на странице *Account page* браузер дает указание подключить устройство безопасности (рис. 7.9).

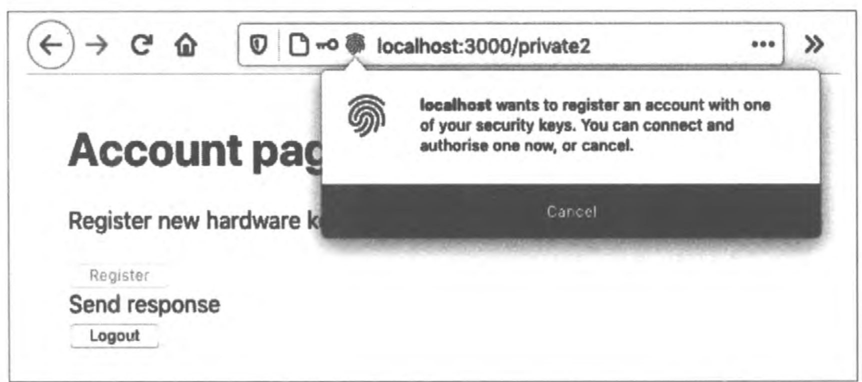


Рис. 7.9. Диалоговое окно с инструкциями по регистрации физического устройства

После подключения и активирования устройства ключа приложение отправляет учетные данные устройства на сервер, а затем выводит сообщение о регистрации нового ключа (рис. 7.10).

Теперь рассмотрим следующий этап — утверждение. Утверждение происходит, когда пользователь подтверждает свою личность при входе в систему.

Шаги этого процесса во многом похожи на шаги для аттестации:

1. Приложение дает указание серверу создать запрос на утверждение.
2. Пользователь преобразует этот запрос в объект утверждения, активируя свое устройство безопасности.

3. Сервер проверяет утверждение на соответствие сохраненным на нем учетным данным, чтобы удостовериться, что данное лицо является тем, кем оно себя называет.

Рис. 7.10. Сообщение о регистрации нового физического ключа

Начнем рассмотрение этого процесса с первого шага, на котором создается запрос на утверждение. В листинге 7.12 приведен код запроса на утверждение.

Листинг 7.12. Код запроса на утверждение

```
{
  "allowCredentials": [
    {"id": "existingTokenID", "type": "public-key"}
  ],
  "attestation": "direct",
  "extensions": {
    "credProps": true,
  },
  "rpID": "localhost",
  "timeout": 60000,
  "challenge": "someRandomString"
}
```

Атрибут `allowCredentials` содержит массив допустимых зарегистрированных устройств. Браузер использует этот массив для проверки, подключил ли пользователь правильное устройство.

Запрос на утверждение также содержит строку `challenge`; это произвольная строка, для которой устройство безопасности должно создать подпись, используя свой секретный ключ. Сервер проверяет действительность этой подписи, используя открытый ключ, чтобы убедиться в том, что подключено корректное устройство.

Значение переменной `timeout` определяет время, в течение которого пользователь будет иметь возможность подтвердить свою личность.

При вызове конечной точки /startVerify с использованием определенного идентификатора пользователя `userID` сервер генерирует запрос на утверждение:

```
import axios from 'axios'
...
//Запрашивает контрольное слово, чтобы проверить userID пользователя
const response = await axios.post('/startVerify', { userID })
```

После этого запрос на утверждение можно передать функции `get()` из библиотеки `webauthn-json`, которая дает пользователю указание подтвердить свою личность, подключив допустимое устройство (листинг 7.13).

Листинг 7.13. Передача запроса на утверждение функции get()

```
import { get } from '@github/webauthn-json'
import axios from 'axios'
...
const response = await axios.post('/startVerify', { userID })
const assertion = await get({ publicKey: response.data })
```

Результат исполнения функции `get()` показан на рис. 7.11.

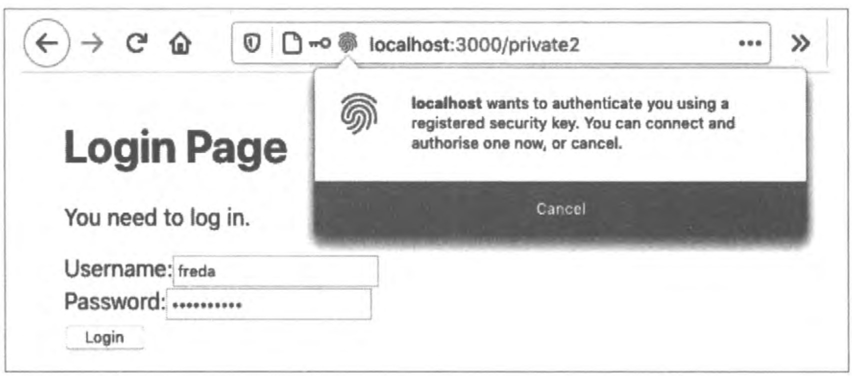


Рис. 7.11. Функция `get()` дает указание пользователю подключить устройство

Функция `get()` возвращает объект утверждения, который содержит подпись для контрольной строки `challenge`, отправленной обратно конечной точке /verify сервера для проверки подписи. Ответ на вызов этой функции сообщит нам, подтвердил ли пользователь свою личность (листинг 7.14).

Листинг 7.14. Подтверждение личности пользователя

```
import { get } from '@github/webauthn-json'
import axios from 'axios'
...
const response = await axios.post('/startVerify', { userID })
const assertion = await get({ publicKey: response.data })
```

```
const resp2 = await axios.post('/verify', { userID, assertion })
if (resp2.data && resp2.data.verified) {
   // Пользователь успешно подтвержден
}
```

Где в приложении можно вставить этот код?

Наш пример приложения основан на рецепте защищенных маршрутов[3]. Он содержит компонент `SecurityProvider`, который управляет обеспечением безопасности всех своих дочерних компонентов. Компонент `SecurityProvider` предоставляет функцию `login`, которая вызывается с параметрами `username` и `password`, когда пользователь отправляет форму входа со своими учетными данными. В этот компонент мы и вставим код верификации, как показано в листинге 7.15.

Листинг 7.15. Вставка кода верификации в компонент SecurityProvider

```
import { useState } from 'react'
import SecurityContext from './SecurityContext'
import { get } from '@github/webauthn-json'
import axios from 'axios'
const SecurityProvider = (props) => {
   const [loggedIn, setLoggedIn] = useState(false)
   return (
      <SecurityContext.Provider
         value={{
            login: async (username, password) => {
               const response = await axios.post('/login', {
                  username,
                  password,
               })
               const { data } = response
               if (data.twoFactorNeeded) {
                  const userID = data.userID
                     const response = await axios.post('/startVerify', {
                     userID,
                  })
                  const assertion = await get({ publicKey: response.data })
                  const resp2 = await axios.post('/verify', {
                     userID,
                     assertion,
                  })
                  if (resp2.data && resp2.data.verified) {
                     setLoggedIn(true)
                  }
```

[3] См. *раздел 2.6*.

```
            } else {
               setLoggedIn(true)
            }
         },
         logout: async () => {
            await axios.post('/logout')
            setLoggedIn(false)
         },
         loggedIn,
      }}
   >
      {props.children}
   </SecurityContext.Provider>
  )
}
export default SecurityProvider
```

Имя пользователя `username` и пароль `password` отправляются конечной точке /login. Если для пользователя есть зарегистрированное устройство безопасности, то в ответе на вызов /login значению атрибута `twoFactorNeeded` будет присвоено значение `true`. Затем вызываем конечную точку /startVerify, передавая ей в качестве параметра идентификатор `ID` пользователя, и используем полученный запрос на утверждение, чтобы дать пользователю указание активировать свое устройство. Полученное утверждение отправляется обратно на сервер. Если нет никаких ошибок, присваиваем переменной `loggedIn` значение `true` и отображаем защищенную страницу.

Рассмотрим весь этот процесс в действии. Предположим, что мы уже зарегистрировали физический ключ по нашей учетной записи. Открываем приложение и щелкаем по ссылке **Account page** (рис. 7.12).

Рис. 7.12. На домашней странице приложения щелкаем по ссылке **Account page**

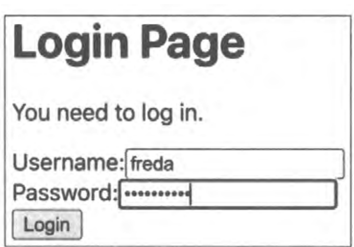


Рис. 7.13. Форма для входа на закрытую страницу

Страница **Account page** закрытая, поэтому нам нужно запросить имя пользователя и пароль (рис. 7.13). В примере приложения имя пользователя **`freda`**, а пароль — **`password`**.

После ввода имени пользователя и пароля и нажатия кнопки **Login** на странице **Login Page** браузер дает указание подключить устройство безопасности (рис. 7.14).

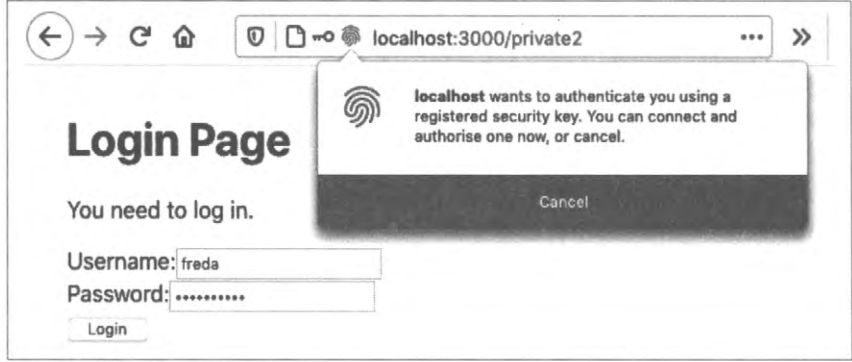


Рис. 7.14. Браузер дает пользователю указание активировать его устройство безопасности

После подключения и активирования устройства безопасности пользователь получает доступ к закрытой странице (рис. 7.15).

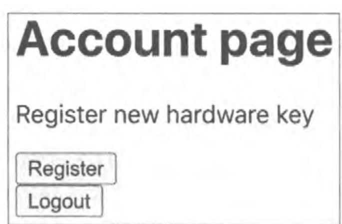


Рис. 7.15. После подтверждения своей личности пользователь получает доступ к странице **Account page**

Обсуждение

Как вы, наверное, уже поняли, API-интерфейс WebAuthn довольно сложный. В нем используется довольно неясная терминология (аттестация вместо регистрации и утверждение вместо верификации), а также несколько низкоуровневых типов данных, работать с которыми, к счастью, не придется благодаря библиотеке `webauthn-json`.

Вся сложная часть находится на сервере. Сервер примера приложения задействует библиотеку SimpleWebAuthn для обработки большей части криптографической составляющей. Если вы планируете применять библиотеку SimpleWebAuthn для серверной части своего приложения, будет полезно знать о наличии клиентской библиотеки SimpleWebAuthn, работающей совместно с серверной. В данном примере мы не использовали эту библиотеку в исходном коде клиента, чтобы не делать код слишком привязанным к ней.

При реализации двухфакторной аутентификации нужно подумать о том, что делать в случае, если пользователь потеряет свое устройство безопасности. С технической точки зрения все, что нужно сделать, чтобы реактивировать его учетную запись, это удалить устройство, зарегистрированное под его учетной записью. Но лучше всего быть предельно осторожным, поскольку типичный взлом двухфакторной

аутентификации злоумышленник осуществляет, позвонив в службу поддержи от имени владельца физического ключа и сообщив о его потере.

Поэтому нужно будет создать достаточно строгую процедуру, в процессе которой выполняется удостоверение личности любого лица, запрашивающего сброс учетной записи.

Исходный код данного рецепта можно загрузить на веб-сайте GitHub по адресу **https://oreil.ly/Diy5D**.

7.3. Работа с протоколом HTTPS

ЗАДАЧА

Протокол HTTPS наиболее часто встречается в рабочей среде, но иногда бывают обстоятельства, когда может быть полезным использовать этот протокол в процессе разработки. Например, некоторые сетевые службы могут работать только из страниц, защищенных протоколом HTTPS. А технология WebAuthn может работать удаленно только через протокол HTTPS[4]. Если приложению требуется прокси-сервер с HTTPS, то его код может постепенно заполниться ошибками и возникнут проблемы.

В настоящее время разрешение протокола HTTPS на рабочих серверах сравнительно прямолинейная задача[5], но как это сделать на сервере для разработки?

РЕШЕНИЕ

Для приложений React, созданных посредством `create-react-app`, протокол HTTPS можно разрешить следующим образом:

- создав самозаверенный SSL-сертификат;
- зарегистрировав этот сертификат с используемым сервером для разработки.

Чтобы создать самозаверенный сертификат, нужны некоторые знания о том, как работает протокол HTTPS.

Протокол HTTPS — это просто протокол HTTP, пропускаемый через туннель зашифрованного SSL-подключения (Secure Sockets Layer — протокол защищенных сокетов). Когда браузер подключается к сайту с HTTPS-адресом, он открывает соединение с защищенным сокетом на сервере[6]. Сервер должен предоставить сертификат, изданный организацией, которой браузер доверяет. Если браузер примет этот сертификат, он отправит зашифрованные данные на защищенный сокет на сервере, которые будут там расшифрованы и переданы далее серверу HTTP.

---

[4] Эту проблему можно обойти на устройствах Android, подключая их через компьютер для разработки. См. *раздел 7.7*.

[5] Дополнительную информацию см. на веб-сайте Let's Encrypt (**https://letsencrypt.org**).

[6] По умолчанию это будет на порту 443.

Главная трудность с организацией сервера HTTPS состоит в получении сертификата, которому браузер будет доверять. Браузер хранит набор корневых сертификатов, изданных крупными надежными организациями. Когда HTTPS-сервер предоставляет браузеру сертификат, этот сертификат должен быть подписан одним из корневых сертификатов браузера. Чтобы создать SSL-сертификат, сначала нужно сформировать корневой сертификат и дать указание браузеру доверять этому сертификату. Затем нужно создать сертификат для нашего сервера разработки, подписав его созданным ранее корневым сертификатом.

Если это выглядит сложно и запутанно, то значит, так оно и есть.

Начнем с создания корневого сертификата. Для этого на вашем компьютере нужно будет установить криптографическую библиотеку OpenSSL.

Чтобы создать файл секретного ключа, нужно исполнить команду `openssl`. После этого необходимо дважды ввести парольную фразу (листинг 7.16).

Листинг 7.16. Создание файла секретного ключа

```
$ openssl genrsa -des3 -out mykey.key 2048
Generating RSA private key, 2048 bit long modulus
......................................................+++
................................+++
e is 65537 (0x10001)
Enter pass phrase for mykey.key:
Verifying - Enter pass phrase for mykey.key:
$
```

Созданный файл mykey.key содержит секретный ключ, который можно использовать для шифрования данных. А файл ключа позволяет создать файл сертификата, содержащий информацию об издавшей его организации и конечной дате его действия.

Файл сертификата создается, как показано в листинге 7.17.

Листинг 7.17. Создание файла сертификата

```
$ openssl req -x509 -new -nodes -key mykey.key -sha256 -days 2048
-out mypem.pem
Enter pass phrase for mykey.key:
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
Введите фразу пароля для файла mykey.key:
Далее нужно будет ввести информацию, которая будет вставлена
в ваш запрос сертификата.
```

```
То, что вам нужно будет ввести, называется различимое имя (или РИ).
Будет несколько полей ввода, но некоторые из них можно оставить пустыми
Некоторые из полей ввода будут содержать значение по умолчанию
Если в поле ввести '.', то оно будет оставлено пустым
-----
Country Name (2 letter code) []:US
State or Province Name (full name) []:Massachusetts
Locality Name (eg, city) []:Cambridge
Organization Name (eg, company) []:O'Reilly Media
Organizational Unit Name (eg, section) []:Harmless scribes
Common Name (eg, fully qualified host name) []:Local
Email Address []:me@example.com
$
```

В данном случае мы создаем сертификат, который будет действителен в течение следующих 2048 дней. Запрашиваемая здесь парольная фраза — это фраза, введенная при создании файла mykey.key. Данные об организации не имеют значения, поскольку мы будем применять этот сертификат только на нашей локальной машине.

Сертификат хранится в файле mypem.pem; этот файл нужно будет установить в качестве корневого сертификата на вашем компьютере[7]. Установить корневой сертификат на компьютере можно несколькими способами[8]. Корневой сертификат можно использовать для подписания сертификатов веб-сайтов, чем мы и займемся далее.

Сначала создадим файл локального ключа и *файл CSR* (Certificate Signing Request — запрос на подписание сертификата), как показано в листинге 7.18.

Листинг 7.18. Создание файла локального ключа и файла CSR

```
$ openssl req -new -sha256 -nodes -out myprivate.csr -newkey rsa:2048 \
-keyout myprivate.key \
-subj "/C=US/ST=Massachusetts/L=Cambridge/O=O'Reilly \
Media/OU=Harmless scribes/CN=Local/emailAddress=me@example.com"
Generating a 2048 bit RSA private key
....................+++
..+++
writing new private key to 'myprivate.key'
-----
$
```

[7] Расширение *pem* означает Privacy-Enhanced Mail — электронная почта повышенной секретности. Формат PEM изначально предназначался для использования с электронной почтой, но в настоящее время служит в качестве общего формата для хранения сертификатов.

[8] Подробная информация на эту тему предоставляется в руководстве компании BounCA (**https://oreil.ly/9NN1H**).

Далее создадим файл extfile.txt и вставим в него содержимое, приведенное в листинге 7.19.

Листинг 7.19. Содержимое файла extfile.txt

```
authorityKeyIdentifier=keyid,issuer
basicConstraints=CA:FALSE
keyUsage=digitalSignature,nonRepudiation,keyEncipherment,dataEncipherment
subjectAltName=DNS:localhost
```

Наконец, исполним команду, которая создаст SSL-сертификат для нашего приложения (листинг 7.20).

Листинг 7.20. Создание сертификата для приложения

```
$ openssl x509 -req -in myprivate.csr -CA mypem.pem -CAkey mykey.key \
-CAcreateserial -out \
myprivate.crt -days 500 -sha256 -extfile ./extfile.txt
Signature ok
subject=/C=US/ST=Massachusetts/L=Cambridge/O=O'Reilly
Media/OU=Harmless scribes/CN=Local/
emailAddress=me@example.com
Getting CA Private Key
Enter pass phrase for mykey.key:
$
```

Помните, что указываемая здесь парольная фраза `pass phrase` — это фраза, введенная при создании файла mykey.key.

В результате мы получим два файла, при помощи которых сможем обезопасить наш сервер для разработки:

- myprivate.crt — сертификат, подписанный корневым сертификатом, который убеждает браузер, что тот может доверять нашему приложению;
- myprivate.key — секретный ключ, который будет использоваться для шифрования соединений между сервером для разработки и браузером.

Для приложений, созданных при помощи средства `create-react-app`, протокол HTTPS можно разрешить, добавив следующий код в файл *.evn в папке приложения:

```
HTTPS=true
SSL_CRT_FILE=myprivate.crt
SSL_KEY_FILE=myprivate.key
```

Если теперь перезапустить сервер, то приложение будет доступно по адресу **https://localhost:3000 вместо http://localhost:3000**.

Обсуждение

Задача создания самозаверенных сертификатов довольно сложна, но в некоторых ситуациях без таких сертификатов не обойтись. Но даже если вам и не требуется протокол HTTPS для вашей среды разработки, все же может быть полезным разобраться, что собой представляет этот протокол, как он работает и почему ему можно доверять.

Исходный код данного рецепта можно загрузить на веб-сайте GitHub по адресу **https://oreil.ly/BAKAE**.

7.4. Аутентификация посредством отпечатка пальца

ЗАДАЧА

В *разделе 7.2* мы рассмотрели, как можно использовать для двухфакторной аутентификации физические ключи, например устройства YubiKeys. Но в настоящее время физические ключи встречаются все еще сравнительно редко и могут быть довольно дорогостоящими. С другой стороны, большинство людей уже обладают мобильными устройствами, такими как сотовые телефоны или планшеты, многие из которых оснащены встроенными сканерами отпечатков пальцев. Теперь выясним, как можно реализовать двухфакторную аутентификацию в приложении React с помощью сканера отпечатков пальцев.

РЕШЕНИЕ

Данную задачу можно решить, применяя сканер отпечатка пальцев как ключ аутентификации WebAuthn. Он подключается к API-интерфейсу таким же образом, хотя требует изменения нескольких конфигурационных настроек.

Данный рецепт основан на материале из *раздела 7.2*. Как вы узнали из этого раздела, процесс аутентификации с использованием API-интерфейса стандарта WebAuthn включает два основных процесса:

- аттестация — пользователь регистрирует физическое устройство по своей учетной записи. Одним из способов аттестации может быть сканирование отпечатка пальца на смартфоне пользователя;
- утверждение — пользователь активирует свое устройство безопасности, или физический ключ, а сервер проверяет, что оно соответствует ранее зарегистрированному устройству или ключу.

Как аттестация, так и утверждение состоит из трех этапов:

- Сервер генерирует запрос на аттестацию или утверждение.
- Пользователь активирует ключ, с помощью которого формируется ответ.
- Ответ отправляется на сервер.

Чтобы переключиться со съемного физического ключа на встроенный в мобильное устройство сканер отпечатка пальца, нужно изменить только поток запроса на аттестацию. В запросе на аттестацию указывается тип ключа, который браузер может зарегистрировать для пользователя. Для рецепта с использованием съемного физического ключа, такого как YubiKeys, мы создали запрос на аттестацию, который выглядел, как показано в листинге 7.21.

Листинг 7.21. Запрос на аттестацию для ключа YubiKeys

```
{
   "rpName": "Physical Token Server",
   "rpID": "localhost",
   "userID": "1234",
   "userName": "freda",
   "excludeCredentials": [
      {"id": "existingKey1", "type": "public-key"}
   ],
   "authenticatorSelection": {
      "userVerification": "discouraged"
   },
   "extensions": {
      "credProps": true,
   },
   }
```

Этот же запрос пригоден и со сканером отпечатков пальцев, нужно лишь слегка модифицировать его, как показано в листинге 7.22.

Листинг 7.22. Запрос на аттестацию для сканера отпечатков пальцев

```
{
   "rpName": "Physical Token Server",
   "rpID": "localhost",
   "userID": "1234",
   "userName": "freda",
   "excludeCredentials": [
   {"id": "existingKey1", "type": "public-key"}
   ],
   "authenticatorSelection": {
   "authenticatorAttachment": "platform",
   "userVerification": "required"
   },
   "attestation": "direct",
   "extensions": {
   "credProps": true,
   },
}
```

За исключением пары изменений, эти два запроса на аттестацию практически одинаковы. Первое изменение касается выбора аутентификатора. Теперь мы хотим задействовать аутентификатор `platform`, поскольку сканер отпечатков пальцев встроен в устройство и не поддается отсоединению. Это означает, что мы фактически даем возможность пользователю работать исключительно с текущим физическим устройством. В противоположность, физический ключ YubiKey можно отсоединить от одного устройства и подключить к другому.

Мы также указываем, что необходимо реализовать прямую аттестацию. Это означает, что нам не потребуется дополнительная верификация. Например, после сканирования отпечатка пальца пользователю не нужно будет вводить ПИН.

В целом, за исключением изменения исходного объекта запроса на аттестацию, весь остальной код остается таким же. Когда пользователь отвечает на запрос на аттестацию, прижав палец к сканеру отпечатков пальцев, сканер сгенерирует открытый ключ, который будет сохранен по учетной записи пользователя. При последующем входе пользователя в систему и подтверждении своей личности сканированием отпечатка пальца, сканер создаст контрольную строку таким же образом, как это сделал бы ключ YubiKey.

Таким образом, если планируется поддержка одного типа аутентификатора, целесообразно позволить пользователям применять как сканер отпечатков пальцев, так и съемные физические ключи.

Если только съемный физический ключ не работает также и с мобильными устройствами, например посредством NFC-связи, маловероятно, что какой-либо пользователь зарегистрирует как съемный физический ключ, так и отпечаток пальца. Зарегистрировав отпечаток пальца, он не сможет войти в систему и зарегистрировать съемный физический ключ, и наоборот.

В листинге 7.23 приведен модифицированный код компонента для регистрации физического ключа, позволяющий регистрировать отпечаток пальца.

Листинг 7.23. Код компонента, модифицированный для регистрации отпечатка пальца

```
import { useState } from 'react'
import Logout from './Logout'
import axios from 'axios'
import { create } from '@github/webauthn-json'
const Private2 = () => {
   const [busy, setBusy] = useState(false)
   const [message, setMessage] = useState()
   const registerToken = async (startRegistrationEndpoint) => {
      setBusy(true)
      try {
         const response = await axios.post(startRegistrationEndpoint)
         setMessage('Send response')
         const attestation = await create({ publicKey: response.data })
         setMessage('Create attestation')
```

```
        const attestationResponse = await axios.post('/register', {
          attestation,
        })
        setMessage('registered!')
        if (
          attestationResponse.data &&
          attestationResponse.data.verified
        ) {
          alert('New key registered')
        }
      } catch (err) {
        setMessage('' + err)
      } finally {
        setBusy(false)
      }
    }
    return (
      <div className="Private2">
        <h1>Account page</h1>
        {window.PublicKeyCredential && (
          <>
            <p>Register new hardware key</p>
            <button
              onClick={() => registerToken('/startRegister')}
              disabled={busy}
            >
              Register Removable Token
            </button>
            <button
              onClick={() => registerToken('/startFingerprint')}
              disabled={busy}
            >
             Register Fingerprint
            </button>
          </>
        )}
        <div className="Account-message">{message}</div>
        <Logout />
      </div>
    )
}
export default Private2
```

Код листинга 7.23 отличается от исходного тем, что для регистрации отпечатка пальца вызывается другая концевая точка. Во всех остальных отношениях код остается прежним.

Чтобы испытать этот код в действии, требуется наличие устройства со сканером отпечатков пальцев. При исполнении приложения на локальном компьютере или удаленном сервере через протокол HTTPS допустим только стандарт WebAuthn. Для тестирования этого кода с мобильным устройством необходимо настроить протокол HTTPS на сервере разработки (см. *раздел 7.3*), или же на устройстве нужно будет настроить прокси-сервер, чтобы перенаправлять подключения к *localhost* на ваш сервер для разработки (см. *раздел 7.7*).

Для запуска примера приложения, нужно перейти в его папку и запустить сервер для разработки:

```
$ npm run start
```

Также необходимо запустить API-сервер. Для этого нужно открыть отдельный терминал, перейти в подпапку сервера и запустить его:

```
$ cd server
$ npm run start
```

Сервер для разработки будет исполняться на порту 3000, а API-сервер — на порту 5000. Сервер для разработки будет перенаправлять API-запросы API-серверу.

Открываем приложение и щелкаем по ссылке **Account page** (рис. 7.16).

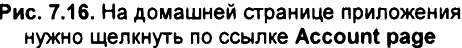


Рис. 7.16. На домашней странице приложения нужно щелкнуть по ссылке **Account page**

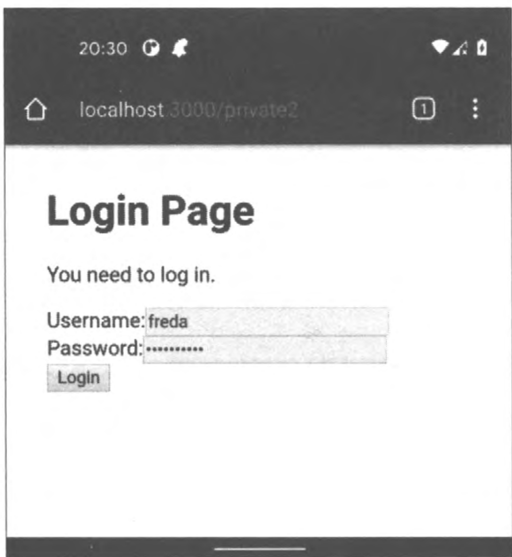


Рис. 7.17. На форме входа в систему введите имя пользователя freda и пароль mypassword

В результате откроется страница входа в систему, на которой нужно ввести имя пользователя `freda` и пароль `mypassword` в соответствующие поля (рис. 7.17). Эти значения жестко закодированы в сервере примера.

Откроется страница **Account page**, содержащая две кнопки для регистрации ключей по вашей учетной записи: одна кнопка для съемных физических ключей (**Register**

Removable Token), а другая — для регистрации отпечатка пальца (**Register Fingerprint**) (рис. 7.18).

Нажмите кнопку для регистрации отпечатка пальца. Мобильное устройство отобразит указание прижать палец к сканеру отпечатков пальцев. После этого сканер сгенерирует открытый ключ, который будет сохранен по учетной записи пользователя, в данном случае `freda`. После успешной регистрации отпечатка пальца откроется окно **Account page** с соответствующим сообщением, как показано на рис. 7.19.

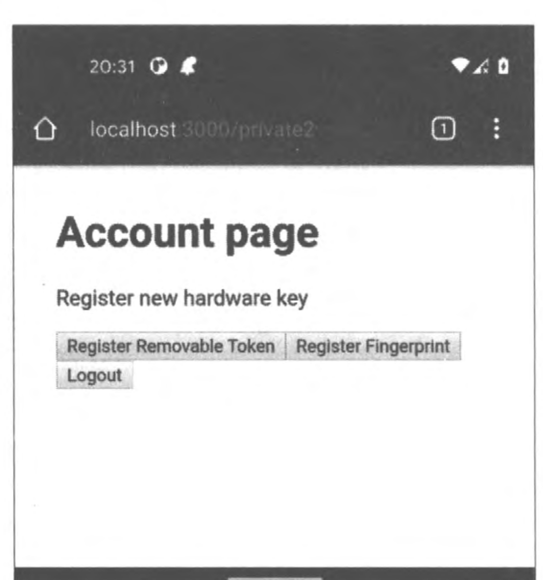


Рис. 7.18. Страница для регистрации съемных физических ключей или отпечатка пальца

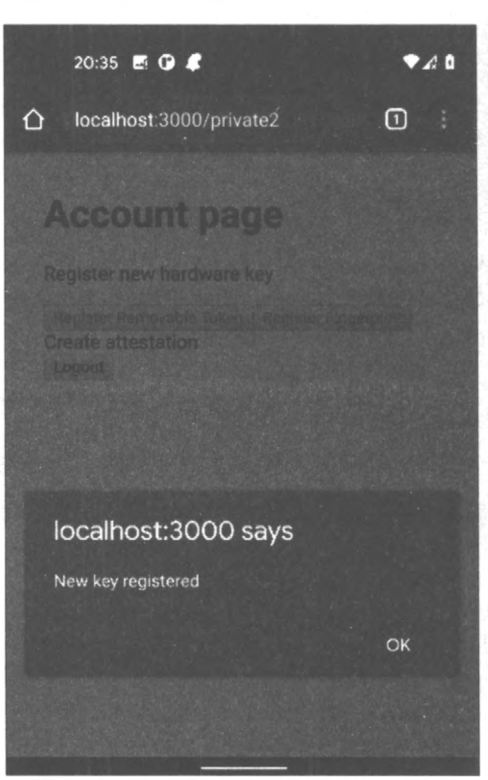


Рис. 7.19. Подтверждение успешной регистрации отпечатка пальца

Теперь выполните выход из системы, а затем снова войдите по учетной записи `freda` и паролю `mypassword`. Приложение выведет сообщение с указанием подтвердить вашу личность, прижав палец к сканеру отпечатков пальцев, и после успешного подтверждения позволит войти обратно в систему.

Обсуждение

Встроенные сканеры отпечатков пальцев встречаются намного чаще, чем съемные физические ключи наподобие ключей YubiKeys, и особенности использования этих двух средств безопасности отличаются. Ключи YubiKeys можно переносить с одно-

го устройства на другое, тогда как отпечаток пальца обычно привязан только к одному устройству[9]. Таким образом, съемные физические ключи предоставляют дополнительную гибкость для пользователей, которым, возможно, нужно подключаться с нескольких устройств. А отрицательный аспект съемных ключей состоит в том, что их намного легче потерять, чем мобильный телефон. В большинстве случаев целесообразно поддерживать оба типа ключей, оставляя за пользователями решение, какой из них подходит им больше.

Исходный код данного рецепта можно загрузить на веб-сайте GitHub по адресу **https://oreil.ly/m8hs6**.

7.5. Подтверждение действий, предоставляя учетные данные

ЗАДАЧА

Иногда пользователь может захотеть выполнить более опасные, чем обычно, или трудно обратимые действия. Например, он может захотеть уничтожить какие-либо важные данные, удалить учетную запись какого-либо пользователя или отправить сообщение электронной почты. Таким пользователем может оказаться и злоумышленник, получивший доступ к оставленному без присмотра компьютеру, в который был выполнен вход. Как можно в таком случае не допустить выполнение вредительских действий?

РЕШЕНИЕ

Многие системы заставляют пользователей подтвердить свою личность вводом своих учетных данных, прежде чем позволить им выполнять опасные операции. В случаях, когда такое подтверждение необходимо выполнять для нескольких разных операций, полезно иметь возможность осуществлять все эти подтверждения централизованно.

Для данного рецепта мы возьмем за основу код для безопасных маршрутов из *раздела 2.6*. В том разделе мы создали компонент `SecurityProvider`, предоставляющий функции входа и выхода из системы для своих дочерних компонентов (листинг 7.24).

Листинг 7.24. Код компонента `SecurityProvider`

```
import { useState } from 'react'
import SecurityContext from './SecurityContext'
const SecurityProvider = (props) => {
   const [loggedIn, setLoggedIn] = useState(false)
   return (
      <SecurityContext.Provider
```

[9] Исключением может быть подключение с внешним сканером отпечатков пальцев.

```
         value={{
            login: (username, password) => {
               // Примечание для группы разработки:
               // Возможно, здесь нужен более высокий уровень безопасности...
               if (username === 'fred' && password === 'password') {
                  setLoggedIn(true)
               }
            },
            logout: () => setLoggedIn(false),
            loggedIn,
         }}
      >
         {props.children}
      </SecurityContext.Provider>
   )
}
export default SecurityProvider
```

Компоненты, которым нужно использовать функции входа и выхода из системы, могут получить доступ к ним из хука `useSecurity`:

```
const security = useSecurity()
...
// В любом месте, где нужно выполнить выход...
security.logout()
```

Для этого рецепта мы добавим в компонент `SecurityProvider` дополнительную функцию, которая будет позволять дочерним элементам подтверждать факт входа пользователя. После предоставления правильных учетных данных пользователю будет позволено выполнять опасную операцию.

Можно создать функцию, которая принимает функцию обратного вызова, содержащую опасную операцию, которая вызывается приложением после того, как пользователь введет правильные учетные данные. Эту функцию будет легче реализовать в компоненте `SecurityProvider`, но с ней будут некоторые проблемы, если вызывать ее из какого-либо другого компонента. По результатам ее исполнения можно было бы возвращать флаг успеха или неуспеха. Код вызова функции показан в листинге 7.25.

Листинг 7.25. Вызов функции для выполнения опасной операции

```
// Мы НЕ БУДЕМ использовать этот подход
confirmLogin((success) => {
   if (success) {
      // Сюда вставляется код для выполнения опасной операции
   } else {
      // Сюда идет код для обработки отмены проверки учетных данных
   }
})
```

Недостаток описанного подхода состоит в том, что если забыть проверить значение флага `success`, то код выполнит опасную операцию, даже при отмене пользователем операции проверки учетных данных.

Альтернативно, можно передавать две отдельные функции обратного вызова: одну для успешной верификации учетных данных и еще одну для отмены операции (листинг 7.26).

Листинг 7.26. Передача двух функций обратного вызова

```
// Мы НЕ БУДЕМ использовать этот подход
confirmLogin(
   () => {
      // Сюда вставляется код для выполнения опасной операции
   },
   () => {
      // Сюда идет код для обработки отмены проверки учетных данных
   });
```

Но этот код слегка уродлив.

Поэтому мы реализуем подход с учетом обещания, что сам компонент будет сложнее, зато упростится любой код, вызывающий его.

В листинге 7.27 приведен окончательный код компонента `SecurityProvider`, содержащий последний вариант функции верификации учетных данных `confirmLogin`.

Листинг 7.27. Полный код компонента SecurityProvider

```
import { useRef, useState } from 'react'
import SecurityContext from './SecurityContext'
import LoginForm from './LoginForm'
export default (props) => {
   const [showLogin, setShowLogin] = useState(false)
   const [loggedIn, setLoggedIn] = useState(false)
   const resolver = useRef()
   const rejecter = useRef()
   const onLogin = async (username, password) => {
      // Заметка для группы разработки:
      // Возможно, здесь нужен более высокий уровень безопасности...
      if (username === 'fred' && password === 'password') {
         setLoggedIn(true)
      }
   }
   const onConfirmLogin = async (username, password) => {
      // Примечание для группы разработки:
      // Возможно, здесь нужен более высокий уровень безопасности...
      return username === 'fred' && password === 'password'
   }
```

```
  return (
    <SecurityContext.Provider
      value={{
        login: onLogin,
        confirmLogin: async (callback) => {
          setShowLogin(true)
          return new Promise((res, rej) => {
            resolver.current = res
            rejecter.current = rej
          })
        },
        logout: () => setLoggedIn(false),
        loggedIn,
      }}
    >
      {showLogin ? (
        <LoginForm
          onLogin={async (username, password) => {
            const valid = await onConfirmLogin(username, password)
            if (valid) {
              setShowLogin(false)
              resolver.current()
            }
          }}
          onCancel={() => {
            setShowLogin(false)
            rejecter.current()
          }}
        />
      ) : null}
      {props.children}
    </SecurityContext.Provider>
  )
}
```

При вызове функции `confirmLogin` компонент `SecurityProvider` отображает форму для ввода учетных данных, чтобы пользователь мог подтвердить свою личность. Функция `confirmLogin` возвращает обещание, которое будет улажено только при вводе пользователем правильных имени пользователя и пароля. При отмене пользователем операции проверки учетных данных обещание отклоняется.

Здесь мы не приводим подробный код компонента `LoginForm`, но его можно найти в исходном коде для этого рецепта.

Код в данном примере проверяет правильность имени пользователя и пароля, сверяя их со статическими строками. Но в рабочей версии приложения эту проверку нужно заменить вызовом какой-либо службы безопасности.

При вызове функции `confirmLogin` обещание сохраняется в ссылке *ref*. Ссылки *ref* обычно указывают на элементы в модели DOM, но их можно использовать для хранения любой части состояния. В отличие от хука `useState`, ссылки *ref* обновляются немедленно. В общем, использование в коде большого количества ссылок *ref* не приветствуется, поэтому мы используем их здесь лишь для того, чтобы иметь возможность немедленно сохранить обещание, без необходимости ожидать завершения исполнения хука `useState`.

Но как использовать функцию `confirmLogin` на практике? Предположим, что у нас есть компонент, содержащий кнопку, которая выполняет некую опасную операцию (листинг 7.28).

Листинг 7.28. Компонент с кнопкой, выполняющей опасную операцию

```
import { useState } from 'react'
import Logout from './Logout'
const Private1 = () => {
   const [message, setMessage] = useState()
   const doDangerousThing = () => {
      setMessage('DANGEROUS ACTION!')
   }
   return (
      <div className="Private1">
         <h1>Private page 1</h1>
         <button
            onClick={() => {
               doDangerousThing()
            }}
         >
            Do dangerous thing
         </button>
         <p className="message">{message}</p>
         <Logout />
      </div>
   )
}
export default Private1
```

Если мы хотим, чтобы для выполнения этой операции пользователь подтвердил свои учетные данные, то можем сначала получить контекст, предоставляемый компонентом `SecurityProvider`:

```
const security = useSecurity()
```

Затем в коде, выполняющем опасную операцию, мы можем ожидать обещание, возвращаемое функцией `confirmLogin`:

```
const security = useSecurity()
...
await security.confirmLogin()
setMessage('DANGEROUS ACTION!')
```

Код, следующий за функцией `confirmLogin`, будет исполняться только при условии предоставления пользователем правильных учетных данных.

Если пользователь закроет диалоговое окно ввода учетных данных, обещание будет отклонено, и эту отмену можно обработать в блоке `catch`.

В листинге 7.29 приведена модифицированная версия компонента, выполняющего опасную операцию, который теперь требует подтвердить личность пользователя, прежде чем выполнять эту операцию.

Листинг 7.29. Код компонента, требующий верификации учетных данных

```
import { useState } from 'react'
import Logout from './Logout'
import useSecurity from './useSecurity'
export default () => {
   const security = useSecurity()
   const [message, setMessage] = useState()
   const doDangerousThing = async () => {
      try {
         await security.confirmLogin()
            setMessage('DANGEROUS ACTION!')
      } catch (err) {
         setMessage('DANGEROUS ACTION CANCELLED!')
      }
   }
   return (
      <div className="Private1">
         <h1>Private page 1</h1>
         <button
            onClick={() => {
               doDangerousThing()
            }}
         >
            Do dangerous thing
         </button>

         <p className="message">{message}</p>

         <Logout />
      </div>
   )
}
```

Чтобы испытать этот код, нужно сначала запустить приложение из его папки:

```
$ npm run start
```

В открывшейся домашней странице приложения (рис. 7.20) щелкните по ссылке **Private Page 1**.

В ответ приложение отобразит указание ввести учетные данные (рис. 7.21) вместе с соответствующими полями. Введите имя пользователя `fred` и пароль `password`.

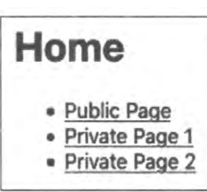


Рис. 7.20. На домашней странице приложения щелкните по ссылке **Private Page 1**

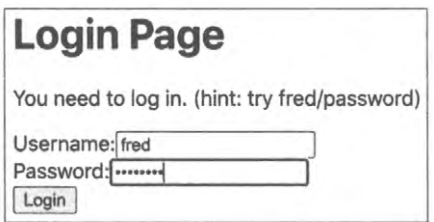


Рис. 7.21. Страница закрытая, поэтому нужно верифицировать личность пользователя

Если теперь нажать кнопку для выполнения опасной операции, то прежде чем будет разрешено выполнить эту операцию, нужно снова верифицировать личность пользователя, введя соответствующие учетные данные (рис. 7.22).

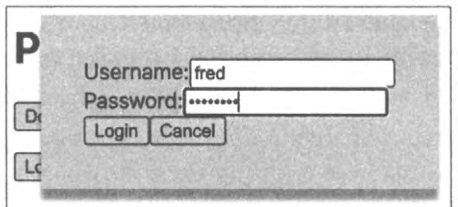


Рис. 7.22. Чтобы выполнить опасную операцию, нужно ввести свои учетные данные

Обсуждение

В данном рецепте код для верификации учетных данных находится в одном центральном месте, а именно в компоненте `SecurityProvider`. Связанная с этим выгода состоит не только в уменьшении объема кода в компонентах, но также в получении возможности выполнять верификацию учетных данных пользователя в специализированных хуках. Если абстрагировать набор операций в определенную службу на основе хука[10], то в эту службу можно также включить и верификационную логику. В результате наши компоненты не будут ничего знать о том, какие операции являются опасными, а какие — нет.

Исходный код данного рецепта можно загрузить на веб-сайте GitHub по адресу **https://oreil.ly/zP75q**.

[10] Пример такой службы можно посмотреть в хуке `useForum` в *разделе 5.2*.

7.6. Однофакторная аутентификация

ЗАДАЧА

Мы уже видели, как съемные физические ключи и отпечатки пальцев позволяют повысить безопасность учетной записи пользователя в системе двухфакторной аутентификации.

Но эти два способа обеспечения дополнительной безопасности можно использовать просто как удобный способ входа в систему. Многие мобильные приложения позволяют выполнить вход лишь одним сканированием отпечатка пальца, без необходимости ввода имени пользователя и пароля.

Но как можно разрешить однофакторную аутентификацию в приложении React?

РЕШЕНИЕ

Ключи безопасности, такие как отпечатки пальцев и USB-устройства YubiKeys, нужно регистрировать на сервере по учетной записи их пользователя. Проблема с использованием этих ключей для однофакторной аутентификации состоит в том, что мы не знаем, кем должен быть пользователь, чей отпечаток пальца был отсканирован. В двухфакторной системе он предварительно идентифицирует себя вводом своего имени пользователя и пароля, но в однофакторной системе нам нужно знать, кто это при создании запроса на утверждение[11].

Проблемы можно избежать, сохраняя в браузере файл cookie, содержащий идентификатор пользователя, при входе в систему пользователя с учетной записью с поддержкой физических ключей[12].

При отображении формы для ввода учетных данных приложение может проверить наличие файла cookie, а затем учесть его при создании запроса на утверждение и запросить у пользователя ключ безопасности. Если пользователь не желает использовать ключ, он может отменить запрос и просто ввести учетные данные в форму[13].

В качестве идентификаторов пользователя часто применяются внутренние ключи, сгенерированные компьютером, которые не содержат никакой закрытой информации. Но если идентификатор содержит данные, позволяющие определить личность пользователя, например адрес электронной почты, то тогда этот подход не следует использовать.

---

[11] Запрос на утверждение требуется тогда, когда браузер дает указание пользователю отсканировать отпечаток пальца или активировать физический ключ. Запрос содержит список разрешенных устройств и поэтому будет уникальным для каждого данного пользователя.

[12] Вследствие такого подхода пользователь сможет выполнять однофакторную аутентификацию только на том браузере, на котором он уже зарегистрировал свой ключ. При работе на другом браузере или после недавней очистки файлов cookie ему придется возвращаться к входу по учетным данным.

[13] Это предполагает наличие файла cookie, читаемого посредством JavaScript. Но пригоден также файл cookie HTTP-типа, читаемый только сервером (или сервис-воркером). В последнем случае на сервере нужно иметь код для проверки, должен ли пользователь предоставлять физический ключ.

Для данного рецепта мы возьмем за основу код для безопасных маршрутов из *раздела 2.6*. Все управление безопасностью будем осуществлять посредством компонента-оболочки `SecurityProvider`, который будет обеспечивать дочерние компоненты функциями входа в систему и выхода из нее. Мы добавим в него еще одну функцию — `loginWithToken`.

Листинг 7.30. Добавление функции `loginWithToken` в компонент `SecurityProvider`

```
import { useState } from 'react'
import SecurityContext from './SecurityContext'
import { get } from '@github/webauthn-json'
import axios from 'axios'
const SecurityProvider = (props) => {
   const [loggedIn, setLoggedIn] = useState(false)
   return (
      <SecurityContext.Provider
         value={{
            login: async (username, password) => {
               const response = await axios.post('/login', {
                  username,
                  password,
               })
               setLoggedIn(true)
            },
            loginWithToken: async (userID) => {
               const response = await axios.post('/startVerify', {
                  userID,
               })
               const assertion = await get({ publicKey: response.data })
               await axios.post('/verify', { userID, assertion })
               setLoggedIn(true)
            },
            logout: async () => {
               await axios.post('/logout')
               setLoggedIn(false)
            },
            loggedIn,
         }}
      >
         {props.children}
      </SecurityContext.Provider>
   )
}
export default SecurityProvider
```

Функция `loginWithToken` принимает в качестве аргумента идентификатор `userID` пользователя, а затем выводит указание пользователю верифицировать свою личность посредством физического ключа следующим образом:

1. Вызывая функцию `startVerify` на сервере, чтобы создать запрос на утверждение.
2. Передавая запрос интерфейсу WebAuthn, чтобы дать указание пользователю прижать свой палец к сканеру отпечатков пальцев.
3. Передавая созданное утверждение обратно концевой точке `verify`, чтобы проверить действительность физического ключа.

В своей реализации приложения вам нужно будет заменить конечные точки для `startVerify` и `verify` их версиями, соответствующими требованиям вашего приложения.

Чтобы в компоненте `SecurityProvider` вызвать функцию `loginWithToken`, нам нужно найти в файлах cookie идентификатор текущего пользователя. Для этого сначала нужно установить библиотеку `js-cookie`:

```
$ npm install js-cookie
```

Библиотека позволит считывать файл cookie для `userID` следующим образом:

```
import Cookies from 'js-cookie'
...
const userIDCookie = Cookies.get('userID')
```

Теперь мы можем использовать этот код в компоненте `Login`, который будет проверять наличие файла cookie для идентификатора `userID`. При наличии такого файла, пользователю будет предложено выполнить вход посредством физического ключа. В противном случае пользователю будет предоставлена возможность выполнить вход посредством ввода имени пользователя и пароля. В листинге 7.31 приведен код окончательной версии приложения, содержащий все эти модификации.

Листинг 7.31. Код окончательной версии приложения

```
import { useEffect, useState } from 'react'
import useSecurity from './useSecurity'
import Cookies from 'js-cookie'
const Login = () => {
   const { login, loginWithToken } = useSecurity()
   const [username, setUsername] = useState()
   const [password, setPassword] = useState()
   const userIDCookie = Cookies.get('userID')
   useEffect(() => {
      ;(async () => {
         if (userIDCookie) {
            loginWithToken(userIDCookie)
         }
      })()
   }, [userIDCookie])

   return (
      <div>
```

```
        <h1>Login Page</h1>
        <p>You need to log in.</p>

        <label htmlFor="username">Username:</label>
        <input
          id="username"
          name="username"
          type="text"
          value={username}
          onChange={(evt) => setUsername(evt.target.value)}
        />
        <br />
        <label htmlFor="password">Password:</label>
        <input
          id="password"
          name="password"
          type="password"
          value={password}
          onChange={(evt) => setPassword(evt.target.value)}
        />

        <br />
        <button onClick={() => login(username, password)}>Login</button>
      </div>
  )
}
export default Login
```

Теперь можно испытать наше приложение в действии. Сначала перейдем в папку приложения и запустим сервер разработки:

```
$ npm run start
```

Затем откроем другое окно терминала и запустим API-сервер:

```
$ cd server
$ npm run start
```

Сервер для разработки будет исполняться на порту 3000, а API-сервер — на порту 5000.

Открываем приложение и щелкаем по ссылке Private Page 1 (рис. 7.23).

Рис. 7.23. На домашней странице приложения щелкаем по ссылке **Private Page 1**

В открывшейся странице входа вводим имя пользователя `freda` и пароль `mypassword` в соответствующие поля (рис. 7.24)

В отрывшейся странице **Account page** предоставляется выбор возможности входа с использованием сканера отпечатков пальцев или физического ключа (рис. 7.25). Регистрируем один из этих вариантов и выходим из системы.

При следующем входе в систему будет сразу же предложено выполнить вход при помощи ранее зарегистрированного физического ключа (рис. 7.26).

Рис. 7.24. Выполняем вход в систему, введя соответствующие учетные данные

Рис. 7.25. Выбор физического ключа для регистрации

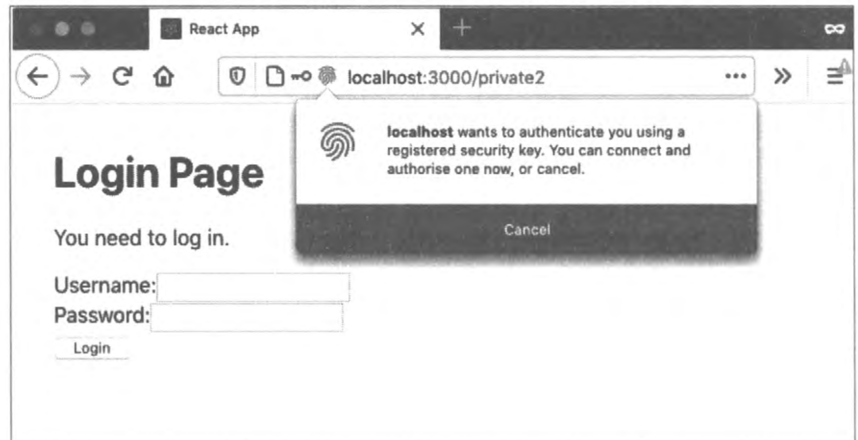


Рис. 7.26. После регистрации физического ключа последующие входы в систему можно выполнять посредством этого ключа

Если теперь активировать физический ключ, то вход в систему будет выполнен без предоставления учетных данных.

Обсуждение

Важно отметить, что однофакторная аутентификация повышает уровень удобства, а не безопасности. Особенно удобны в этом отношении сканеры отпечатков пальцев, поскольку с ними для входа нужно буквально лишь шевельнуть одним пальцем.

Но при этом всегда необходимо обеспечить возможность запасного входа посредством ввода учетных данных. Это не понизит уровень безопасности вашего прило-

жения, поскольку какой-либо злоумышленник может удалить файл cookie и использовать форму ввода учетных данных в любом случае.

Исходный код данного рецепта можно загрузить на веб-сайте GitHub по адресу **https://oreil.ly/4ZDh6**.

7.7. Проверка приложения на устройстве Android

ЗАДАЧА

Многие операции тестирования мобильных приложений можно выполнять в браузере персонального компьютера, эмулирующего внешний вид мобильного устройства (рис. 7.27).

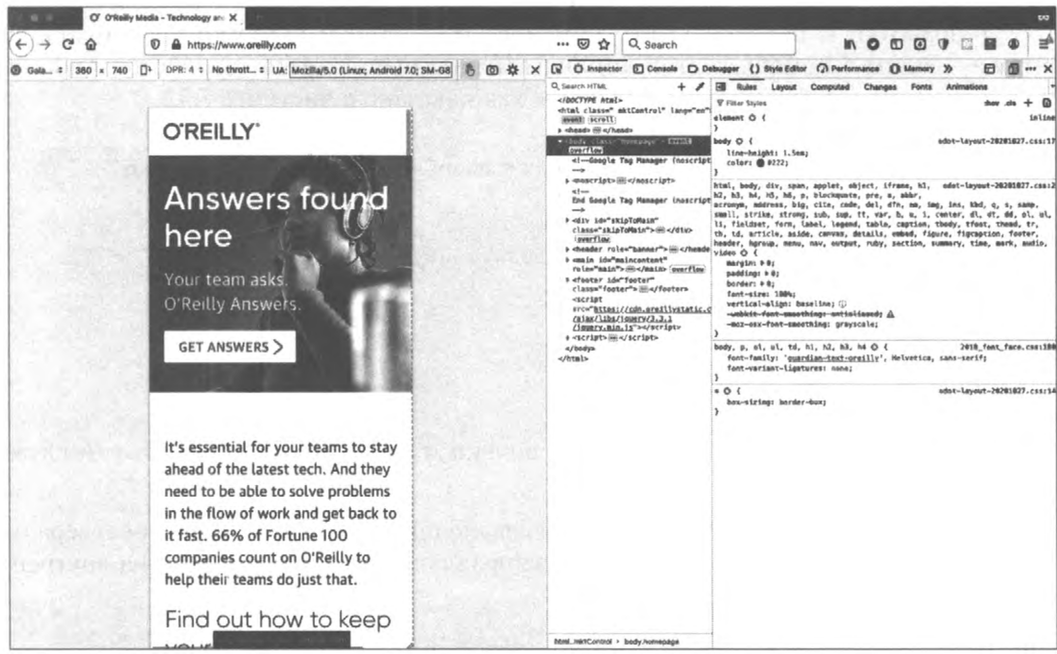


Рис. 7.27. Приложения для мобильных устройств можно тестировать в браузере персональных компьютеров

Но иногда тестирование приложений React лучше всего выполнять на физическом мобильном устройстве. Обычно это не представляет никаких проблем, поскольку мобильное устройство может получить удаленный доступ к приложению React через IP-адрес компьютера разработки.

Но такое тестирование может быть невозможным при некоторых обстоятельствах:

- мобильное устройство не может подключиться к сети компьютера разработки;
- используется технология, например WebAuthn, требующая применения протокола HTTPS для всех доменов, кроме *localhost*.

Можно ли настроить мобильное устройство для доступа к приложению React, как будто бы оно исполнялось на *localhost*, даже если оно исполняется на отдельном компьютере?

РЕШЕНИЕ

В этом рецепте мы рассмотрим, как организовать прокси-сервис на устройстве Android, чтобы подключения к *localhost* направлялись на сервер компьютера разработки.

Первое, что нам потребуется, — это устройство Android, на котором разрешена отладка через USB (**https://oreil.ly/fc5Fv**). Также нужно установить на своем компьютере набор разработчика Android SDK (**https://oreil.ly/BFeXr**), чтобы получить доступ к инструменту Android Debug Bridge (ADB). Инструмент ADB открывает канал связи между компьютером разработки и устройством Android.

Далее при помощи кабеля USB нужно подключить устройство Android к компьютеру разработки и проверить, что команда adb доступна в терминале командной строки[14]. Затем с помощью этой команды можно просмотреть устройства Android, подключенные к компьютеру разработки, как показано в листинге 7.32.

Листинг 7.32. Перечисление подключенных к компьютеру устройств Android

```
$ adb devices
* daemon not running; starting now at tcp:5037
* daemon started successfully
List of devices attached
25PRIFFEJZWWDFWO device
$
```

В данном случае к компьютеру подключено только одно устройство Android с идентификатором 25PRIFFEJZWWDFWO.

Далее, опять же посредством команды adb, выполняем настройку прокси-сервера на устройстве Android, который будет перенаправлять весь трафик HTTP на внутренний порт устройства 3000:

```
$ adb shell settings put global http_proxy localhost:3000
```

Если к компьютеру подключено больше одного устройства Android, требуемое устройство нужно выбрать, указав его идентификатор посредством опции команды adb — -s<идентификатор-устройства>.

Далее запускаем на устройстве Android прокси-службу, которая будет перенаправлять весь трафик с порта 3000 устройства на порт 3000 компьютера разработки:

```
$ adb reverse tcp:3000 tcp:3000
```

[14] Для этого нужно найти на вашем компьютере папку, в которой установлен набор разработчика Android SDK. Команда adb будет находиться в одной из подпапок этой папки.

Если теперь на устройстве Android открыть браузер и ввести в строку адреса адрес **http://localhost:3000**, то в браузере отобразится приложение, исполняющееся на компьютере разработки, как будто бы оно исполнялось на данном устройстве Android (рис. 7.28).

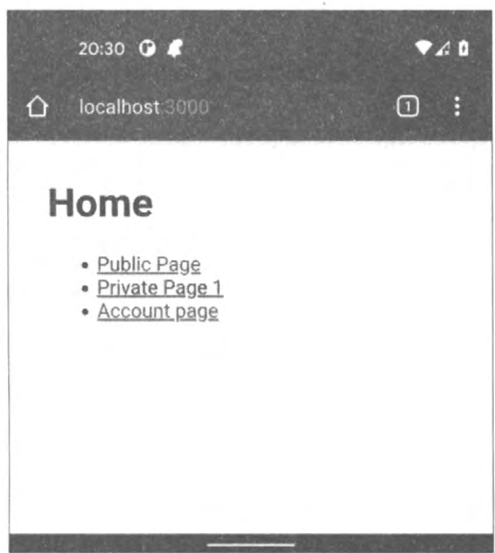


Рис. 7.28. В браузере устройства Android по адресу **localhost** отображается приложение React, исполняющееся на компьютере разработки

Закончив работать с приложением, нужно отключить настройки прокси-сервера на устройстве Android, задав для прокси-сервера порт `:0`:

```
$ adb shell settings put global http_proxy :0
```

Если этого не сделать, то устройство не будет иметь доступа к сети.

Обсуждение

Для этого рецепта нужно выполнить большой объем подготовительной работы, поскольку он требует установки на компьютере разработки всего набора разработчика Android SDK. Но затем подключение и отключение физических устройств Android к компьютеру не будет представлять никаких проблем.

7.8. Проверка безопасности посредством ESlint

ЗАДАЧА

В коде JavaScript угрозу безопасности чаще всего представляют лишь несколько распространенных ошибок кодирования. С этой угрозой можно бороться, разработав набор стандартов кодирования, чтобы не допускать этих ошибок. Но, чтобы

поддерживать соответствие данных стандартов последним изменениям в технологии, их придется часто пересматривать. Кроме того, потребуются также трудоемкие и дорогостоящие экспертизы.

Можно ли каким-либо образом выполнять проверку на несоблюдение должных приемов безопасного кодирования, не замедляя при этом процесс разработки?

РЕШЕНИЕ

Одним из подходов к внедрению проверок безопасности может быть их автоматизация. Для этого можно использовать один из соответствующих инструментов, например линтер (linter — средство контроля качества кода) ESlint. Если вы создали свое приложение при помощи средства `create-reat-app`, то, скорее всего, линтер ESlint уже установлен на вашем компьютере. Более того, средство `create-react-app` исполняет ESlint при каждом запуске своего сервера для разработки. Если вам когда-либо приходилось видеть в терминале сообщения об ошибках в коде, то эти сообщения были сгенерированы средством ESlint. В листинге 7.33 приведен пример таких сообщений.

Листинг 7.33. Пример сообщений от ESlint

```
Compiled with warnings.
(Скомпилировано с предупреждениями.)
src/App.js
Line 5:9: 'x' is assigned a value but never used no-unused-vars
(Строка 5:9: Переменной 'x' присвоено значение, но переменная никогда
не использовалась. Неиспользуемые переменные не допускаются.)
Search for the keywords to learn more about each warning.
(Дополнительную информацию по каждому предупреждению можно получить, выполнив поиск
по ключевым словам.)
To ignore, add // eslint-disable-next-line to the line before.
Чтобы игнорировать предупреждение, добавьте // eslint-disable-next-line
в предшествующую строку
```

Если средство ESlint еще не установлено на вашем компьютере, его можно установить, выполнив следующую команду `npm`:

```
$ npm install --save-dev eslint
```

После установки средства, его нужно инициализировать, как показано в листинге 7.34.

Листинг 7.34. Инициализация средства ESlint

```
$ node_modules/.bin/eslint --init
- How would you like to use ESLint? · problems
(Как вы хотите использовать ESlint? · Обнаружение проблем)
```

```
- What type of modules does your project use? · esm
- (Какие типы модулей используются в вашем проекте? · esm)
- Which framework does your project use? · react
(Какой фреймворк используется в вашем проекте?  · react)
- Does your project use TypeScript? · No / Yes
(Используется ли вашем проекте TypeScript? · No / Yes)
- Where does your code run? · browser
(Где исполняется ваш код? · В браузере)
- What format do you want your config file to be in? · JavaScript
(Какой формат файла конфигурации вы хотите использовать? · JavaScript)
Local ESLint installation not found.
(Локальная установка ESlint не обнаружена.)
The config that you've selected requires the following dependencies:
(Для выбранной вами конфигурации требуются следующие зависимости:)
eslint-plugin-react@latest eslint@latest
- Would you like to install them now with npm? · No / Yes
(Хотите ли вы установить их сейчас посредством npm? · No / Yes)
$
```

Но при этом следует иметь в виду, что если вы используете средство `create-react-app`, то инициализировать ESlint не требуется, поскольку инициализация уже была выполнена.

На данном этапе можно создать собственный набор правил для ESlint, чтобы проверять на несоблюдение любых правил безопасности. Но будет намного легче установить подключаемый модуль ESlint, уже содержащий готовый набор правил.

Давайте, например, установим пакет eslint-plugin-react-security, созданный и поддерживаемый компанией Slyk (**https://slyk.io**):

```
$ npm install --save-dev eslint-plugin-react-security
```

После установки модуля его нужно активировать, отредактировав раздел `eslintConfig` файла package.json (если используется средство `create-react-app`) или файла *eslintrc** в папке приложения.

Подлежащий редактированию код приведен в листинге 7.35.

Листинг 7.35. Код, подлежащий редактированию, для активирования модуля

```
"eslintConfig": {
   "extends": [
      "react-app",
      "react-app/jest"
   ]
},
```

Этот фрагмент кода нужно заменить кодом, приведенным в листинге 7.36.

Листинг 7.36. Новый код для активирования модуля

```
"eslintConfig": {
   "extends": [
      "react-app",
      "react-app/jest"
   ],
   "plugins": [
      "react-security"
   ],
   "rules": {
      "react-security/no-javascript-urls": "warn",
      "react-security/no-dangerously-set-innerhtml": "warn",
      "react-security/no-find-dom-node": "warn",
      "react-security/no-refs": "warn"
   }
},
```

В результате этих изменений будут активированы четыре правила из модуля React Security.

Чтобы проверить их в работе, добавим в приложение немного кода, который нарушает правило `no-dangerously-set-innerhtml` (листинг 7.37).

Листинг 7.37. Код, нарушающий правило `no-dangerously-set-innerhtml`

```
import logo from './logo.svg'
import './App.css'
function App() {
   return (
      <div className="App">
         <header className="App-header">
            <img src={logo} className="App-logo" alt="logo" />
            <p>
               Edit <code>src/App.js</code> and save to reload.
            </p>
            <div
               dangerouslySetInnerHTML={{
                  __html: '<p>This is a bad idea</p>',
               }}
            />
            <a
               className="App-link"
               href="https://reactjs.org"
               target="_blank"
               rel="noopener noreferrer"
            >
```

```
                Learn React
            </a>
        </header>
    </div>
  )
}
export default App
```

Если средство ESlint было установлено вручную, этот файл можно отсканировать следующей командой:

```
$ node_modules/.bin/eslint src/App.js
```

А если используется средство `create-react-com`, то нужно перезапустить сервер, чтобы перезагрузить конфигурацию ESlint. Результаты сканирования приведены в листинге 7.38.

Листинг 7.38. Предупреждение, выдаваемое средством ESlint

```
Compiled with warnings.
src/App.js
   Line 12:16: dangrouslySetInnerHTML prop usage detected
               react-security/no-dangerously-set-innerhtml
Search for the keywords to learn more about each warning.
To ignore, add // eslint-disable-next-line to the line before.
```

Обсуждение

Проверки ESlint с помощью хука `pre-commit` Git полезно выполнять и в случае работы с командой разработчиков, чтобы не допустить сдачи разработчиками кода, который не способен пройти проверку. Это позволит быстрее получать оценку работы и не допустит вывода из строя всей сборки из-за ошибки одного разработчика.

Для настройки хуков `pre-commit` посредством файла package.json рассмотрите возможность установки хуков кода Husky (**https://oreil.ly/uEjix**).

Еще одно преимущество, предоставляемое автоматизацией проверок безопасности, состоит в том, что их можно добавить в используемый вами конвейер сборки и развертывания приложений. Если выполнять проверки в начале конвейера, то коммиты можно отклонять сразу же и сообщать об этом разработчикам.

Исходный код данного рецепта можно загрузить на веб-сайте GitHub по адресу **https://oreil.ly/kvBcS**.

7.9. Удобные для браузера формы входа в систему

ЗАДАЧА

Многие решения безопасности подразумевают наличие формы для ввода имени пользователя и пароля, но при создании таких форм можно легко допустить несколько ошибок, получив результат, неудобный для пользователя. На некоторых устройствах автоматическое исправление ошибок правописания и исправление первой буквы предложения в заглавную может оказать медвежью услугу и исказить имя пользователя и/или пароль. Некоторые браузеры пытаются автоматически завершать вводимое имя пользователя, но часто неясно, согласно каким правилам это делается, в итоге на одних сайтах все работает должным образом, а на других возникают ошибки.

Каких правил нужно придерживаться при создании форм входа в систему, чтобы они органически работали совместно с браузером, не создавая никаких проблем?

РЕШЕНИЕ

Удобство работы с формами входа в систему можно значительно улучшить при помощи нескольких атрибутов HTML.

Первым делом может быть полезным отключить исправление ошибок правописания для полей ввода имени пользователя. Эта возможность часто используется в мобильных устройствах, чтобы компенсировать малый размер клавиатуры и неизбежные вследствие этого ошибки правописания. Но подобное поведение малополезно при вводе имени пользователя. Возможность автоматического исправления ошибок правописания можно отключить при помощи атрибута `autoCorrect` следующим образом:

```
<input autoCorrect="off"/>
```

Далее, если в качестве имени пользователя задан адрес электронной почты, рассмотрите возможность присвоения этому полю типа `email`:

```
<input type="email"/>
```

Это может вызвать запуск специальной клавиатуры для работы с электронной почтой на мобильных устройствах. А некоторые браузеры могут даже отображать список недавних адресов электронной почты в окне автозавершения или в заголовке клавиатуры для электронной почты.

Также рекомендуем рассмотреть возможность указания `j_username` в качестве идентификатора и названия поля ввода имени пользователя.

```
<input id="j_username" name="j_username"/>
```

Почему? Потому что поле с названием `j_username` часто используется в приложениях на основе JavaScript, и поэтому существует вероятность, что пользователь уже

предоставлял значение для такого поля в прошлом. Это, в свою очередь, повышает вероятность того, что браузер может включить адрес электронной почты в окне автозавершения.

Можно также явно указать, что поле является полем ввода имени пользователя, в результате чего повышается *вероятность* активизации автозавершения в браузере:

```
<input autoComplete="username"/>
```

Теперь разберемся с паролями.

Первым делом всегда задавайте тип поля для ввода пароля как `password`:

```
<input type="password"/>
```

Никогда не поддавайтесь искушению воспроизвести внешний вид поля пароля каким-либо иным образом, например через специальное стилистическое CSS-оформление. Это не позволит браузеру применять к полю пароля стандартные средства безопасности, например отключение в нем функции копирования. Кроме того, если не задать полю тип `password`, то браузер не будет предлагать сохранить пароль в своем диспетчере паролей.

Существуют два вида поля пароля: для текущего пароля (при входе в систему) и для нового пароля (при создании или смене пароля).

Почему это важно? Потому что HTML-атрибут `autoComplete` может указывать браузеру предполагаемое назначение поля пароля.

Если оно служит для входа в систему, то нужно указать, что это `current-password`:

```
<input type="password" autoComplete="current-password"/>
```

А если предназначено для создания пароля при регистрации или для изменения пароля, то нужно указать, что это `new-password`:

```
<input type="password" autoComplete="new-password"/>
```

Это значение будет стимулировать браузер к автозавершению поля пароля сохраненным значением пароля, а также активирует любые встроенные или посторонние средства генерирования паролей.

Наконец, избегайте использования форм входа в систему в стиле мастера, т. е. в виде многошагового окна. Пример такого окна (для сайта газеты Washington Post) показан на рис. 7.29.

Это значительно снижает вероятность распознавания браузером назначения одного поля для ввода пароля, в результате чего уменьшается вероятность, что он предложит завершить его сохраненным значением.

Обсуждение

Атрибуту `autoComplete` можно присваивать много других редко используемых значений для разных типов полей, от подробностей адреса и номера телефона до номера кредитной карты. Дополнительная информация по этому вопросу предоставляется на веб-сайте разработчиков Mozilla (**https://oreil.ly/TLHLF**).

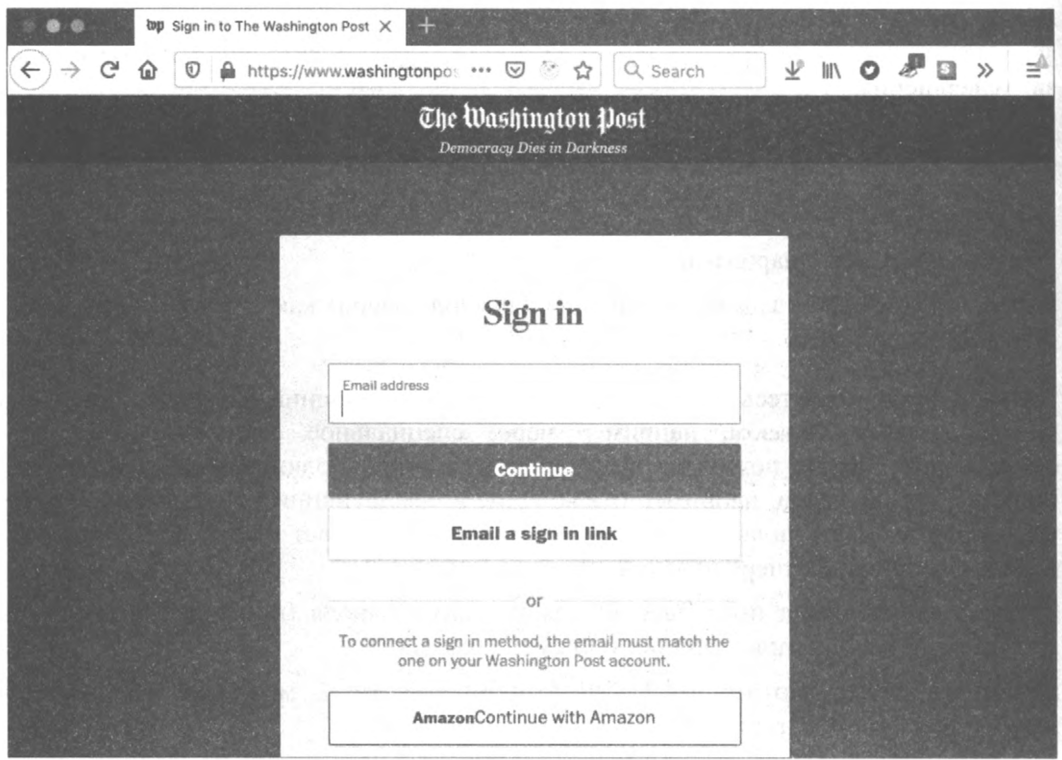


Рис. 7.29. Многошаговая форма входа в систему может помешать браузеру использовать возможность автозавершения

ГЛАВА 8

Тестирование

В этой главе мы рассмотрим разные методы тестирования приложений React. В общем, мы пришли к выводу, что нецелесообразно давать слишком жесткие рекомендации касательно точного состава тестов, которые нужно иметь. Хорошим руководящим принципом будет следовать двум правилам:

- Никогда не приступайте к разработке кода, если у вас нет для него теста, который заведомо обнаружит какую-либо ошибку.
- Если первое исполнение теста оказывается успешным, избавьтесь от него.

Эти два правила помогут вам создавать работающий код, в то же самое время позволяя избежать создания избыточных малоценных тестов.

Мы пришли к выводу, что на ранних этапах проекта лучше создавать тесты на основе браузера. Такие тесты, как правило, более высокого уровня и способствуют определению основных бизнес-требований к приложению. Позже, когда архитектура приложения начинает проявляться и стабилизироваться, становится легче создавать дополнительные модульные тесты для отдельных компонентов. Такие тесты требуют меньше времени для разработки и исполняются быстрее. Кроме того, когда у нас имеется стабильная структура кода, постоянное обновление тестов больше не требуется.

Иногда полезно уточнить само понятие, что именно является тестом. При работе с кодом компоновки страницы, главное значение которого визуальное, в качестве "теста" можно рассматривать рассказ средства Storybook. Проверка выполняется визуально разработчиком, просматривая компонент в процессе его создания. Конечно же, этот вид теста не будет автоматически обнаруживать регрессивные ошибки, но в этом рецепте предоставляется метод, который позволит вам преобразовать такие визуальные тесты в автоматические.

Если создавать тесты, прежде чем приступать к написанию кода программы, то вы обнаружите, что тесты тоже являются инструментами для разработки. Такие тесты станут исполняемыми примерами желаемого поведения приложения.

И наоборот, если разрабатывать тесты после написания кода программы, то они будут просто искусственными объектами, фрагментами кода, которые вы должны слепо создавать, потому что это из разряда вещей, которые должен делать профессиональный разработчик.

В данной главе мы будем концентрироваться на четырех инструментах: библиотеке для тестирования React Testing Library, средстве Storybook, библиотеке Selenium и фреймворке для тестирования Cypress.

Библиотека React Testing Library — замечательное средство для создания очень подробных модульных тестов.

Средство для работы с галереями изображений Storybook уже было рассмотрено ранее в этой книге. Мы включили его в настоящую главу по той причине, что галерея представляет собой набор примеров кода, которыми также являются и тесты. Поэтому вы сможете найти рекомендации, как использовать Storybook в процессе разработки и тестирования.

Библиотека Selenium — одна из наиболее устоявшихся библиотек для тестирования приложений в реальном браузере.

Наконец, фреймворк для тестирования Cypress быстро завоевывает у нас популярность. Этот фреймворк похож на библиотеку Selenium в том отношении, что он исполняется в браузере. Но он содержит большое множество дополнительных возможностей: повторное воспроизведение тестов, генерированные видеопрогонов тестов, а также значительно более простую модель программирования. Если вам придется выбрать только один из этих четырех инструментов, выбирайте фреймворк Cypress.

8.1. Работа с библиотекой React Testing Library

ЗАДАЧА

Существует много способов протестировать приложение React. На ранних стадиях проекта, когда определяется основное назначение и главная функция приложения, желательно создавать тесты очень высокого уровня, например тесты Cucumber (**https://cucumber.io**). При исследовании какой-либо изолированной части системы (например, создания и обслуживание элемента данных) целесообразно создать функциональные тесты, используя инструмент наподобие фреймворка Cypress.

Но при уточнении подробностей создания отдельного элемента будет, наверное, желательным прибегнуть к модульным тестам. Название *модульные тесты* отражает их назначение — тестирование отдельного фрагмента кода как изолированного модуля. Хотя можно спорить, действительно ли название "модульный тест" правильное в случае тестирования компонентов (которые часто содержат подкомпоненты и, таким образом, не изолированы), это название часто употребляется с тестами компонентов, которые можно протестировать вне браузера.

Но как выполнить модульное тестирование компонентов React? В разные периоды времени для этого применялось несколько подходов. Первые модульные тесты полагались на отрисовку компонента в строку HTML. Для этого требовалась минимальная инфраструктура тестирования, но одновременно присутствовали многие недостатки:

- Обработка повторных отрисовок при изменении состояния компонента.
- Создание утверждений на элементах HTML, что тест должен выделять элементы посредством парсинга строки.

♦ Для тестирования взаимодействий пользовательского интерфейса требовалось эмулировать модель событий.

Разработчики очень быстро создали библиотеки для обработки подробностей каждой из этих проблем.

Но в созданных таким образом тестах не хватало реальности, получаемой в браузере. Таким образом, теряются тонкости взаимодействия между виртуальной моделью DOM и моделью браузера DOM. Часто с целью уменьшения уровня сложности тестов отрисовка подкомпонентов не выполнялась.

В результате всего этого для приложений React часто было доступно лишь небольшое количество тестов. Поэтому разработчики переписывали код, перемещая сложную логику в функции JavaScript, которые легко поддавались тестированию. Весь более-менее сложный код требовалось тестировать в настоящем браузере, что замедляло процесс тестирования. В результате разработчики предпочитали тестировать лишь небольшое количество сценариев.

Так как же можно подвергнуть компоненты React реалистичному модульному тестированию, избегая при этом накладных расходов в виде исполнения всего приложения и тестирования его в настоящем браузере?

РЕШЕНИЕ

В библиотеке React Testing Library (разработчик Кент С. Доддс — Kent C. Dodds) предпринимается попытка избежать проблем с предыдущими библиотеками модульного тестирования, предоставляя автономную реализацию модели DOM. В результате тесты способны выполнять отрисовку компонентов DOM в виртуальной модели DOM, которую затем можно синхронизировать с моделью DOM библиотеки тестирования и создать дерево элементов HTML, которые ведут себя так, будто бы они функционируют в настоящем браузере.

Эти элементы можно проверять таким же самым образом, как и в настоящем браузере. Они обладают такими же атрибутами и свойствами. Можно даже передавать нажатия клавиш в поля ввода и заставить их вести себя аналогично полям ввода в настоящем браузере.

Если вы создали свое приложение при помощи средства `create-reat-app`, то, библиотека React Testing Library уже должна быть установлена на вашем компьютере. В противном случае ее можно установить, выполнив следующие команды:

```
$ npm install --save-dev "@testing-library/react"
$ npm install --save-dev "@testing-library/jest-dom"
$ npm install --save-dev "@testing-library/user-event"
```

Эти три раздела библиотеки позволят выполнять модульное тестирование компонентов.

Библиотека Testing Library позволит выполнять отрисовку компонентов, используя реализацию модели DOM в сопутствующей библиотеке Jest Dom (`@testing-library/jest-dom`). А сопутствующая библиотека User Event (`@testing-library/user-

event) упрощает взаимодействие с элементами модели DOM. Эта библиотека осуществляет поддержку щелчков мыши и ввод с клавиатуры в поля ввода компонентов.

Для демонстрации модульного тестирования компонентов нам потребуется тестовое приложение, которое мы будем использовать в большей части этой главы. При запуске приложение дает указание пользователю выполнить простое вычисление, а затем информирует его о правильности ответа (рис. 8.1).

Рис. 8.1. Тестируемое приложение

Основной компонент библиотеки называется App. Для этого компонента создадим модульный тест, поместив его в файле App.test.js (листинг 8.1).

Листинг 8.1. Модульный тест для компонента App

```
describe('App', () => {
it('should tell you when you win', () => {
// Тест должен вывести сообщение о выигрыше
// Given we've rendered the app
// (При условии, что приложение было отрисовано)
// When we enter the correct answer
// (Когда мы вводим правильный ответ)
// Then we are told that we've won
// (Выводится сообщение, что мы выиграли)
})
})
```

Код в листинге 8.1 реализует тест из библиотеки Jest и содержит один сценарий, который тестирует, что компонент App сообщает пользователю, что он выиграл, если тот предоставит правильный ответ. В данном случае собственно тест представлен заставочными комментариями.

Тест начинается с отрисовки компонента App. Это можно сделать, импортировав компонент и передав его функции render библиотеке Testing Library (листинг 8.2).

Листинг 8.2. Код отрисовки компонента App

```
import { render } from '@testing-library/react'
import App from './App'
describe('App', () => {
   it('should tell you when you win', () => {
      // Тест должен вывести сообщение о выигрыше
      // При условии, что приложение было отрисовано
      render(<App />)
      // Когда мы вводим правильный ответ
      // Выводится сообщение, что мы выиграли
   })
})
```

Обратите внимание на то, что мы передаем функции отрисовки `render` настоящий код XML JavaScript, что означает, что при желании мы могли бы протестировать поведение элемента при передаче других наборов свойств.

Для следующей части теста нам необходимо ввести правильный ответ. Для этого нам сначала нужно знать, что является правильным ответом. Предлагаемое пользователю вычисление всегда представляет собой умножение произвольных чисел, поэтому мы можем выполнить захват чисел со страницы, а затем ввести их произведение в поле `Guess`[1].

Нам нужно будет исследовать элементы, создаваемые компонентом `App`. Функция `render` возвращает объект, содержащий эти элементы и набор функций для их фильтрации. Это возвращаемое значение мы заменяем объектом `screen` библиотеки Testing Library.

Объект `screen` можно рассматривать, как содержимое окна браузера. Он позволяет нам находить элементы в странице, чтобы можно было взаимодействовать с ними. Например, найти поле ввода `Guess` можно следующим образом:

```
const input = screen.getByLabelText(/guess:/i)
```

Методы фильтрации в объекте `screen` обычно начинаются с:

- `getBy...` — если известно, что модель DOM содержит единственный экземпляр совпадающего элемента;
- `queryBy...` — если известно что не совпадает ни один или один элемент;
- `getAllBy...` — если известно, что совпадает один или больше элементов (возвращается массив);
- `queryAllBy...` — возвращает массив всех совпадающих элементов или пустой массив при отсутствии совпадений.

---

[1] В других рецептах в этой главе мы увидим, что из теста можно динамически убрать составляющую произвольности и зафиксировать правильный ответ, не захватывая ответ со страницы.

Если эти методы обнаруживают большее или меньшее количество элементов, чем ожидаемое, то они создают исключение. Существуют также асинхронные версии методов `getBy...` и `getAllBy...` — методы `findBy...` и `findAllBy...` соответственно, которые возвращают обещания.

Каждый из этих методов фильтрации может выполнять определенные поиски (табл. 8.1).

Таблица 8.1. Разновидности методов фильтрации

Суффикс метода	Описание вида поиска
`...ByLabelText`	Находит поле по метке
`...ByPlaceHolderText`	Возвращает поле с подстановочным текстом
`...ByText`	Возвращает поле с совпадающим текстом
`...ByDisplayValue`	Возвращает элементы, содержащие совпадающее значение
`...ByAltText`	Возвращает элементы с совпадающим атрибутом `alt`
`...ByTitle`	Возвращает элементы с совпадающим атрибутом `Title`
`...ByRole`	Выполняет поиск по роли ARIA
`...ByTestId`	Выполняет поиск по атрибуту `data-testid`

Существует почти 50 способов поиска элементов в странице. Но, вы, наверное, заметили, что ни один из рассмотренных способов не использует CSS-селектор для отыскания элементов. Это преднамеренный подход. Библиотека Testing Library ограничивает количество способов для нахождения элементов в модели DOM. Например, нельзя выполнять поиск элементов по имени класса, чтобы не ослаблять тест. Имена классов часто служат для косметического стилевого оформления и нередко подвергаются изменениям.

Но для поиска элементов все же можно задействовать селекторы, применяя контейнер, возвращаемый методом `render`:

```
const { container } = render(<App />)
const theInput = container.querySelector('#guess')
```

Но такой подход считается плохой практикой. При работе с библиотекой Testing Library, наверное, лучше всего будет следовать стандартному подходу и искать элементы по их содержимому или роли.

Одно небольшое отступление от этого подхода — использование методов фильтрации `...ByTestId`. При отсутствии практических способов отыскания элемента по его содержимому к релевантному тегу всегда можно добавить атрибут `data-testid`. Это будет полезно для теста, который мы сейчас разрабатываем, т. к. нам нужно найти два числа, отображаемые на странице. Поскольку эти числа генерируются случайным образом, то мы не знаем их значений (рис. 8.2).

Таким образом, мы слегка модифицируем код, добавив в него тестовые идентификаторы (листинг 8.3).

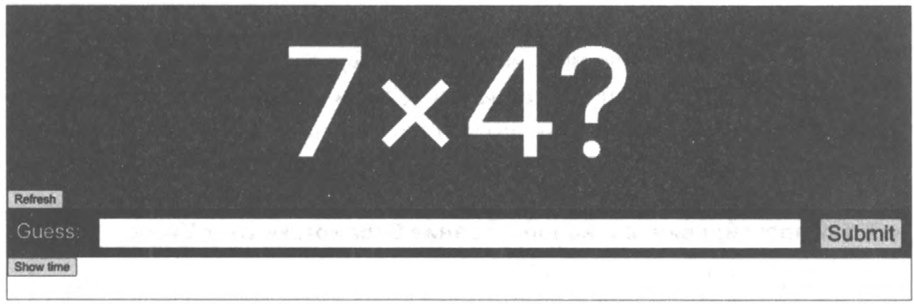


Рис. 8.2. Мы не можем искать числа по их содержимому, поскольку не знаем их значений

Листинг 8.3. Добавление тестовых идентификаторов в код

```
<div className="Question-detail">
   <div data-testid="number1" className="number1">
      {pair && pair[0]}
   </div>
   &times;
   <div data-testid="number2" className="number2">
      {pair && pair[1]}
   </div>
   ?
</div>
```

Это позволит нам реализовать следующую часть нашего теста (листинг 8.4).

Листинг 8.4. Добавление в код следующей части теста

```
import { render, screen } from '@testing-library/react'
import App from './App'
describe('App', () => {
   it('should tell you when you win', () => {
      // Тест должен вывести сообщение о выигрыше
      // При условии, что приложение было отрисовано
      render(<App />)
      // Когда мы вводим правильный ответ
      const number1 = screen.getByTestId('number1').textContent
      const number2 = screen.getByTestId('number2').textContent
      const input = screen.getByLabelText(/guess:/i)
      const submitButton = screen.getByText('Submit')
      // Err...
      // Выводится сообщение, что мы выиграли
   })
})
```

Теперь у нас есть тест для каждого числа, а также есть элемент ввода input. Далее нам нужно ввести правильное число в поле ввода и отправить ответ. Это мы делаем

посредством библиотеки `@testing-library/user-event` (листинг 8.5). Библиотека User Event упрощает процесс создания событий JavaScript для элементов HTML. Библиотека User Event часто импортируется под псевдонимом `user`. Это объясняется тем, что вызовы данной библиотеки можно рассматривать, как выполняемые пользователем действия.

Листинг 8.5. Импортирование и использование библиотеки User Event

```
import { render, screen } from '@testing-library/react'
import user from '@testing-library/user-event'
import App from './App'
describe('App', () => {
   it('should tell you when you win', () => {
      // Тест должен вывести сообщение о выигрыше
      // При условии, что приложение было отрисовано
      render(<App />)
      // При вводе правильного ответа
      const number1 = screen.getByTestId('number1').textContent
      const number2 = screen.getByTestId('number2').textContent
      const input = screen.getByLabelText(/guess:/i)
      const submitButton = screen.getByText('Submit')
      user.type(input, '' + parseFloat(number1) * parseFloat(number2))
      user.click(submitButton)
      // Выводится сообщение, что мы выиграли
   })
})
```

Наконец, нам нужно выдать утверждение, что мы выиграли, т. е. ввели правильный ответ. Это можно сделать, просто выполнив поиск любого элемента, содержащего слово `won`[2]:

```
// Тогда выводится сообщение, что мы выиграли
screen.getByText(/won/i)
```

Это утверждение работает, поскольку метод `getByText` создает исключение, если он не найдет ровно один совпадающий элемент.

Если на каком-либо этапе теста вы не уверены в текущем состоянии HTML, попробуйте добавить в код метод `screen.getByTestId('NONEXISTENT')`. Создаваемое исключение покажет текущий HTML.

Но медленное исполнение приложения может вызвать сбой теста. Это объясняется тем, что методы `get...` и `query...` считывают текущее состояние модели DOM. Если для появления результата требуется пара секунд, то утверждение будет неуспеш-

[2] Обратите внимание на то, что во многих тестах сравнение текста осуществляется посредством регулярных выражений, что позволяет, как в данном примере, выполнять поиск подстрок без различения регистра. Регулярные выражения могут предотвратить частые сбои тестов.

ным. Поэтому желательно обеспечить, чтобы утверждения были асинхронными. Это немного усложнит код, но повысит стабильность теста при работе медленно исполняющейся программы.

Методы `find...` являются асинхронными версиями методов `get...`, а метод `waitFor` библиотеки Testing Library обеспечит повторное исполнение кода в течение некоторого периода времени. Объединив эти два метода, можно создать окончательную версию нашего теста (листинг 8.6).

Листинг 8.6. Окончательная версия теста

```
import { render, screen, waitFor } from '@testing-library/react'
import user from '@testing-library/user-event'
import App from './App'
describe('App', () => {
   it('should tell you when you win', async () => {
      // При условии, что приложение было отрисовано
      render(<App />)
      // При вводе правильного ответа
      const number1 = screen.getByTestId('number1').textContent
      const number2 = screen.getByTestId('number2').textContent
      const input = screen.getByLabelText(/guess:/i)
      const submitButton = screen.getByText('Submit')
      user.type(input, '' + parseFloat(number1) * parseFloat(number2))
      user.click(submitButton)
      // Выводится сообщение, что мы выиграли
      await waitFor(() => screen.findByText(/won/i), { timeout: 4000 })
   })
})
```

Модульные тесты должны исполняться быстро, но если по какой-либо причине тест длится свыше пяти секунд, функции `it` нужно будет передать другое значение `timeout` в миллисекундах.

Обсуждение

Работая с разными командами, мы обнаружили, что на ранних стадиях проекта разработчики пишут модульные тесты для каждого компонента, но со временем число таких тестов начинает постепенно уменьшаться. В конечном итоге они могут даже совсем убрать модульные тесты, если требуют слишком большого объема обслуживания.

Это происходит частично по той причине, что модульные тесты более абстрактные, чем браузерные. Они выполняют то же самое, что и браузерные тесты, но делают это незаметно. Мы не видим их взаимодействия с компонентами.

Вторая причина заключается в том, что разработчики часто воспринимают тесты, просто как подлежащие сдаче составляющие проекта. Команда может даже созда-

вать сборки, которые не будут работать должным образом, без входящих в состав программ модульных тестов.

Эти проблемы обычно устраняются, если разработчики создают тесты, прежде чем приступать к разработке кода. Если сначала создавать тесты, постепенно по одной строке кода за раз, то у вас будет намного лучшее понимание текущего состояния кода HTML. Если перестать рассматривать тесты как составляющие разрабатываемого и начать смотреть на них как на средства для проектирования кода, то из трудоемкого бремени они превратятся в инструменты, облегчающие вашу работу.

При разработке программы важно начинать с теста, гарантированно выявляющего ошибки. На ранних стадиях проекта этим тестом может быть браузерный тест. По мере повышения степени готовности проекта и стабилизации его архитектуры следует создавать все больше модульных тестов.

Исходный код данного рецепта можно загрузить на веб-сайте GitHub по адресу **https://oreil.ly/PITqj**.

8.2. Использование Storybook для тестирования отрисовки

ЗАДАЧА

Тест — это просто пример, который можно исполнить. Соответственно тесты имеют много общего с системами галерей компонентов, например Storybook. Как тесты, так и галереи являются примерами компонентов, исполняющихся в определенных обстоятельствах. Тогда как тест проверяет корректность кода, разработчик делает выводы, взяв библиотечный пример, исследуя его и удостоверяясь, что он выглядит ожидаемым образом. Как в галереях, так и в тестах исключения можно легко увидеть.

Но есть и отличия. Тесты могут автоматически взаимодействовать с компонентами, тогда как галерейные компоненты требуют посредничества пользователя для нажатия кнопок и ввода текста. Тесты можно исполнять одной командой, а галереи нужно просматривать вручную, по одному примеру за раз. Галерейные компоненты визуальные и легко понимаемы, а тесты абстрактны и их создание не доставляет большого удовольствия.

Можно ли каким-либо образом объединить галереи наподобие Storybook с автоматизированными тестами, чтобы в результате получить преимущества одного и другого?

РЕШЕНИЕ

В качестве решения мы рассмотрим, как можно многократно использовать рассказы Storybook в тестах. Установить Storybook в приложение можно следующей командой:

```
$ npx sb init
```

Пример приложения в этой главе прежний — это простая математическая игра, в которой пользователю нужно вычислить ответ на задачу умножения (рис. 8.3).

Один из компонентов игры называется `Question`, он отображает произвольно сгенерированные перемножаемые числа (рис. 8.4).

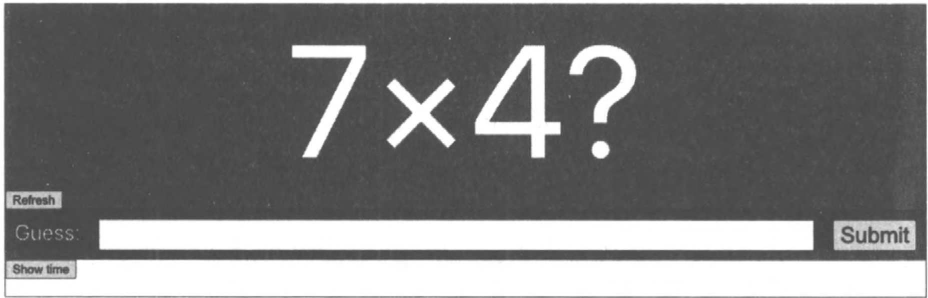


Рис. 8.3. Окно примера приложения

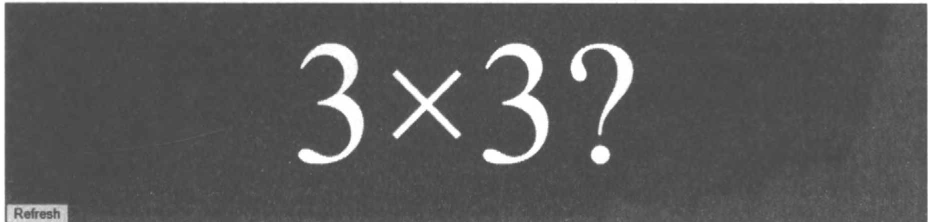


Рис. 8.4. Компонент Question

Предположим, что мы не будем слишком заботиться о тестировании этого компонента. Давайте просто соберем его, создав несколько рассказов Storybook. Для начала создадим файл Question.stories.js (листинг 8.7).

Листинг 8.7. Файл Question.stories.js для компонента `Stories`

```
import Question from './Question'
const Info = {
   title: 'Question',
}
export default Info
export const Basic = () => <Question />
```

Затем создадим начальную версию компонента, который мы можем просмотреть в Storybook и признать его удовлетворительным.

Листинг 8.8. Код компонента `Question`

```
import { useEffect, useState } from 'react'
import './Question.css'
```

```
const RANGE = 10
function rand() {
   return Math.floor(Math.random() * RANGE + 1)
}
const Question = ({ refreshTime }) => {
   const [pair, setPair] = useState()
   const refresh = () => {
      setPair((pair) => {
         return [rand(), rand()]
      })
   }
   useEffect(refresh, [refreshTime])

   return (
      <div className="Question">
         <div className="Question-detail">
            <div data-testid="number1" className="number1">
               {pair && pair[0]}
            </div>
            &times;
            <div data-testid="number2" className="number2">
               {pair && pair[1]}
            </div>
            ?
         </div>
         <button onClick={refresh}>Refresh</button>
      </div>
   )
}
export default Question
```

Данный компонент отображает вопрос с произвольными перемножаемыми числами, когда пользователь нажимает кнопку **Refresh** или когда родительский компонент передает ему новое значение `refreshTime`.

Мы отображаем компонент в Storybook, и похоже, что он работает должным образом. При нажатии кнопки **Refresh** компонент обновляет вопрос. Поэтому на данном этапе мы начинаем использовать этот компонент в основном приложении. Через некоторое время мы добавляем в компонент несколько дополнительных возможностей, но поскольку ни одна из них не вносит визуальных изменений, то мы не проверяем компонент в Storybook снова. Ведь, в конце концов, он будет выглядеть так же, как и раньше. Правильно?

В листинге 8.9 приведена модифицированная версия компонента `Question` в том виде, как он вставляется в основное приложение.

Листинг 8.9. Модифицированная версия компонента

```
import { useEffect, useState } from 'react'
import './Question.css'
const RANGE = 10
function rand() {
   return Math.floor(Math.random() * RANGE + 1)
}
const Question = ({ onAnswer, refreshTime }) => {
   const [pair, setPair] = useState()
   const result = pair && pair[0] * pair[1]
   useEffect(() => {
      onAnswer(result)
   }, [onAnswer, result])

   const refresh = () => {
      setPair((pair) => {
         return [rand(), rand()]
      })
   }
   useEffect(refresh, [refreshTime])

   return (
      <div className="Question">
         <div className="Question-detail">
            <div data-testid="number1" className="number1">
               {pair && pair[0]}
            </div>
            &times;
            <div data-testid="number2" className="number2">
               {pair && pair[1]}
            </div>
            ?
         </div>
         <button onClick={refresh}>Refresh</button>
      </div>
   )
}
export default Question
```

Размер этого фрагмента кода немного больше размера предыдущего. Мы добавили в него функцию обратного вызова `onAnswer`, которая возвращает правильный ответ родительскому компоненту всякий раз, когда приложение генерирует новый вопрос.

Поначалу кажется, что новый компонент работает в приложении должным образом, но затем происходит странная вещь. Когда кто-то решает просмотреть модифици-

рованный компонент в Storybook, он обнаруживает ошибку, как показано на рис. 8.5.

Что здесь происходит? Мы добавили в код неявное допущение, что родительский компонент всегда будет передавать функцию обратного вызова `onAnswer` в наш компонент. Поскольку рассказы Storybook отрисовали рассказ `Basic` без функции `onAnswer`, то мы получили ошибку:

```
<Question/>
```

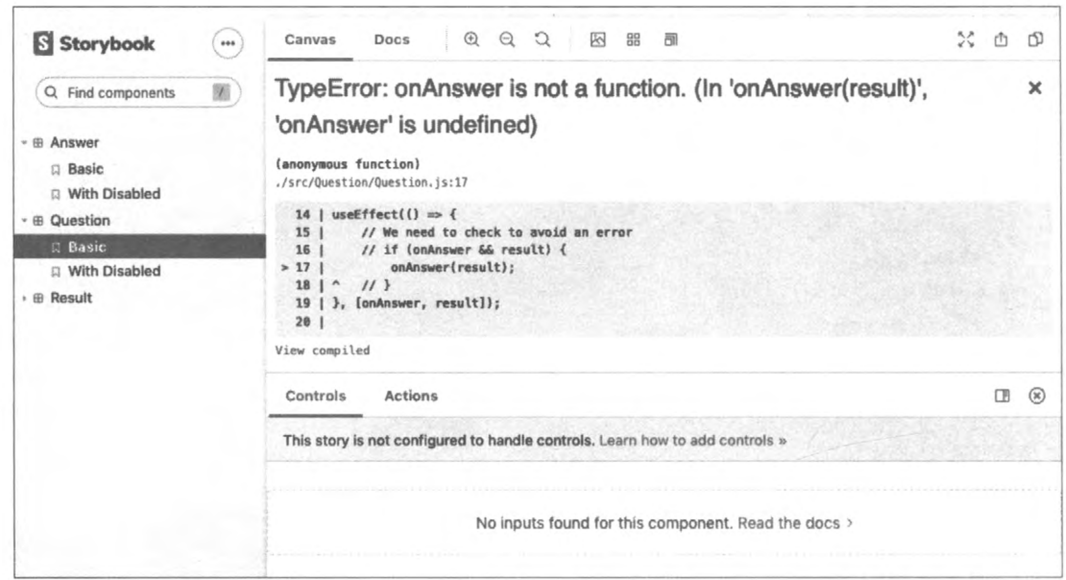


Рис. 8.5. При просмотре новой версии компонента в Storybook обнаруживается ошибка

Важно ли это? Для такого простого компонента, как этот, нет, не важно. В конце концов, само приложение продолжало работать должным образом. Но неспособность справиться с проблемой недостающих свойств, таких как отсутствующая функция обратного вызова в данном случае, или, что случается чаще, отсутствующие данные, является одной из самых типичных причин ошибок в React.

Приложения часто генерируют свойства React, используя сетевые данные, а это означает, что первоначальные свойства, передаваемые компонентам, часто будут неопределенными или иметь значения `null`. Обычно, чтобы избежать подобных проблем, желательно использовать безопасный по отношению к типам язык, например TypeScript, или создавать тесты, которые проверяют способность компонентов справляться с проблемой недостающих свойств.

Мы создали этот компонент без каких бы то ни было тестов, но с рассказом Storybook, который и уловил эту проблему. Так существует ли какой-либо способ создания теста, который бы автоматически проверял, что Storybook может отрисовывать все рассказы?

Теперь создадим тест для этого компонента, который сохраним в файле Question.test.js.

Рассмотрите возможность создания папки для каждого компонента. Вместо того чтобы просто поместить файл Question.js в папке src, создайте папку src/Question и уже в этой папке сохраните файлы Question.js, Question.stories.js и Quesiton.test.js. Если затем добавить файл src/Question/index.js, который выполняет экспорт по умолчанию компонента `Question`, это не повлияет на остальной код, а также будет уменьшено число файлов, с которыми нужно работать другим разработчикам[3].

В файле теста мы можем создать тест Jest, который загружает все рассказы, а затем передает их функции `render` в библиотеку Testing Library (листинг 8.10)[4].

Листинг 8.10. Код теста Jest

```
import { render } from '@testing-library/react'
import Question from './Question'
const stories = require('./Question.stories')
describe('Question', () => {
   it('should render all storybook stories without error', () => {
      for (let story in stories) {
         if (story !== 'default') {
            let C = stories[story]
            render(<C />)
         }
      }
   })
})
```

Если в ваших рассказах есть декораторы для предоставления таких элементов, как маршрутизаторы и стилистическое оформление, они не будут автоматически обнаружены этим методом. Поэтому их нужно добавить в метод `render` в тесте.

Исполнение этого теста выдаст ошибку:

```
onAnswer is not a function
TypeError: onAnswer is not a function
```

Эту ошибку можно исправить, выполняя проверку на наличие функции обратного вызова перед ее вызовом (листинг 8.11).

Листинг 8.11. Проверка на наличие функции обратного вызова

```
useEffect(() => {
   // Нужно выполнить проверку, чтобы избежать ошибки
   if (onAnswer && result) {
      onAnswer(result)
   }
}, [onAnswer, result])
```

---

[3] Подробности о структуре кода примера приложения можно найти в исходном коде, который можно загрузить с репозитория GitHub по адресу **https://oreil.ly/P1Tqj**.

[4] Если у вас еще не установлена библиотека Testing Library, инструкции по ее установке см. в *разделе 8.1*.

Описанный метод позволяет создавать простые тесты компонентов с минимальными усилиями. Целесообразно сначала создать рассказ для компонента, который не содержит вообще никаких свойств. Затем, прежде чем добавлять новое свойство, создайте рассказ, который использует его, и продумайте, каким будет ожидаемое поведение компонента.

Хотя этот тест будет выполнять только простую отрисовку каждого рассказа, нет никаких оснований, препятствующих возможности импортировать один рассказ и создать тест на основе этого рассказа (листинг 8.12).

Листинг 8.12. Код теста на основе рассказа Storybook

```
import { render, screen } from '@testing-library/react'
import user from '@testing-library/user-event'
import Question from './Question'
import { Basic, WithDisabled } from './Question.stories'
...
it('should disable the button when asked', () => {
   render(<WithDisabled />)
   const refreshButton = screen.getByRole('button')
   expect(refreshButton.disabled).toEqual(true)
})
```

Обсуждение

Тесты отрисовки Storybook позволяют внедрить в приложение простое модульное тестирование, которое может обнаружить на удивление большое количество ошибок регрессии. Значит, можно рассматривать тесты как примеры, которые помогают разработчику с проектированием кода, а не как искусственные продукты кодирования, которые необходимо создавать просто, чтобы угодить руководителю группы. Создание тестов отрисовки для рассказов также полезно при работе в команде, не имеющей большого опыта модульного тестирования. Создание визуальных примеров позволяет избежать проблем, которые могут возникнуть вследствие восприятия разработчиками невизуальных тестов как слишком абстрактных. Это также может помочь разработчикам выработать привычку создавать тестовый файл для каждого компонента системы. Тогда при необходимости внести незначительные изменения в компонент будет существенно легче добавить небольшую функцию модульного тестирования, прежде чем модифицировать что-либо.

Исходный код данного рецепта можно загрузить на веб-сайте GitHub по адресу https://oreil.ly/PITqj.

8.3. Тестирование без сервера посредством Cypress

ЗАДАЧА

Одна из главных особенностей высококачественного кода — способ его реагирования на ошибки. Первое из списка восьми заблуждений, составленного Питером Дойчем (Peter Deutsch, The Fallacies of Distributed Computing, **https://oreil.ly/eDtKG**), — сеть является надежной. Ненадежна не только сама сеть, но также и подключаемые к ней серверы и базы данных. Рано или поздно вашему приложению придется иметь дело с какой-либо сетевой неисправностью. Это может быть потеря телефоном сетевого подключения, или выход из строя сервера, или сбой базы данных, или отсутствие данных, которые вы пытаетесь обновить, поскольку их кто-то удалил. Какой бы ни была причина сетевой неисправности, вам потребуется решить, что будет делать ваше приложение, когда эта неисправность возникнет.

Эмулирование сетевых проблем в тестовой среде может оказаться трудной задачей. Код, который переключает сервер в состояние какой-либо ошибки, скорее всего будет создавать проблемы для других тестов или подключающихся к этому серверу пользователей.

Так как же можно создать автоматизированные тесты для случаев сетевых сбоев?

РЕШЕНИЕ

В предлагаемом далее рецепте воспользуемся фреймворком для тестирования Cypress, который мы рассмотрели в *главе 1*. Это по-настоящему выдающаяся тестовая система, которая стремительно становится нашим незаменимым инструментом во многих разработках.

Чтобы установить Cypress в свой проект, выполните следующую команду:

```
$ npm install --save-dev cypress
```

Работа Cypress основана на автоматизации веб-браузера. В этом отношении фреймворк похож на другие системы тестирования, например Selenium. Но все же Cypress отличается тем, что для него не нужно устанавливать отдельный драйвер, и он может как удаленно управлять браузером, так и внедрять себя в движок обработки JavaScript браузера.

Поэтому Cypress может активно заменять основные части инфраструктуры JavaScript эмулированными версиями, которыми он может управлять. Например, Cypress может заменить функцию `fetch` JavaScript, которая предназначена для осуществления сетевых вызовов к серверу[5]. Поэтому тесты Cypress могут имитировать поведение сетевого сервера и позволить разработчику клиентских приложений создавать искусственные ответы сервера.

---

[5] Напрямую или косвенно посредством таких библиотек, как Axios.

Для этого рецепта мы будем пользоваться тем же приложением, что и для других примеров в этой главе. В него мы добавим сетевой вызов для сохранения результатов каждого ответа пользователя на вопрос. Это можно сделать, не создавая настоящего серверного кода, а эмулируя ответ в Cypress.

Чтобы продемонстрировать, как все это работает, мы сначала создадим тест, который эмулирует правильный ответ сервера, а затем тест, эмулирующий сбой сервера.

После установки Cypress создайте в папке cypress/integration файл 0001-basic-game-functions.js и вставьте в него содержимое, приведенное в листинге 8.13[6].

Листинг 8.13. Содержимое файла теста 0001-basic-game-functions.js

```
describe('Basic game functions', () => {
   it('should notify the server if I lose', () => {
      // При условии, что приложение запущено
      // Когда вводится неправильный ответ
      // Сервер сообщает о проигрыше
   })
})
```

На данном этапе фрагменты кода, которые нам нужно будет написать, заменены комментариями-заполнителями.

В Cypress каждая команда и утверждение начинается с префикса `cy`. Отобразить в браузере страницу http://localhost:3000 можно при помощи кода, приведенного в листинге 8.14.

Листинг 8.14. Код для отображения в браузере страницы http://localhost:3000

```
describe('Basic game functions', () => {
   it('should notify the server if I lose', () => {
      // При условии, что приложение запущено
      cy.visit('http://localhost:3000')
      // Когда вводится неправильный ответ
      // Сервер сообщает о проигрыше
   })
})
```

Тест запускается на исполнение следующей командой:

```
$ npx cypress run
```

---

[6] Название файла не имеет значения, но мы следуем соглашению, согласно которому названия подобных тестов начинаются с номера рассказа. Это уменьшает вероятность конфликтов совмещающихся тестов и значительно облегчает отслеживание цели отдельных изменений.

Эта команда исполнит все тесты, не открывая браузера[7]. Исполнять тесты можно и другим образом, посредством следующей команды:

```
$ npx cypress open
```

Данная команда отрывает окно приложения Cypress, как показано на рис. 8.6.

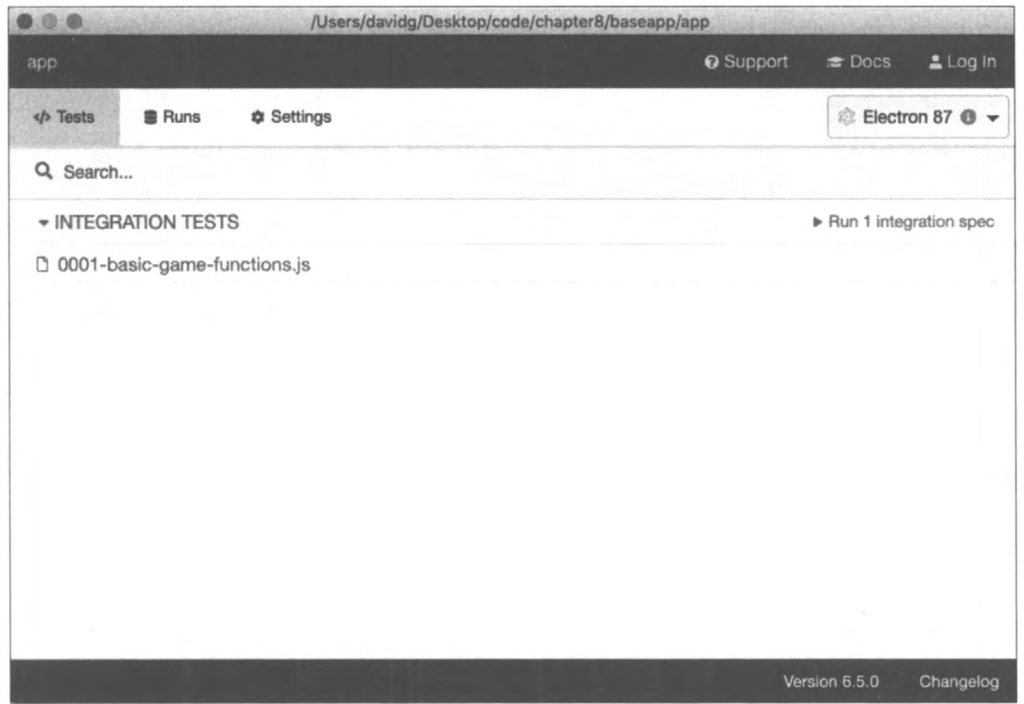


Рис. 8.6. Отображение теста в окне приложения Cypress

Если теперь выполнить двойной щелчок по файлу теста, то он откроется в браузере, как показано на рис. 8.7.

Приложение дает указание пользователю умножить два произвольных числа (рис. 8.8). Предлагаемые числа всегда будут в диапазоне от 1 до 10, и если ввести значение 101, то можно быть уверенным, что это будет неправильный ответ.

Фреймворк Cypress не обеспечивает прямого захвата текстового содержимого с экрана. Поэтому мы не можем просто считать значения этих двух чисел и сохранить их в переменных, поскольку команды в Cypress не выполняют действия в браузере сразу же. Вместо этого Cypress добавляет введенную команду к цепочке команд, которые он исполняет в конце теста. Такой подход может показаться несколько странным, но он позволяет Cypress справляться с большинством проблем, создаваемых асинхронными интерфейсами[8]. Отрицательным аспектом является

[7] Этот способ повышает скорость выполнения тестов и записывает видео для каждого из них, что полезно при исполнении тестов на сервере интеграции.

[8] Команды Cypress во многом похожи на обещания, хотя таковыми не являются. Их можно рассматривать как "полу-обещания".

то, что никакая команда не может возвратить содержимое страницы, поскольку при исполнении команды страницы уже не будет.

В других местах этой главы мы рассмотрим, как можно убрать случайность из тестовых сценариев и сделать этот тест детерминированным, что устранит необходимость захвата данных со страницы.

Найти поле ввода по селектору CSS можно при помощи команды `cy.get`. А кнопку **Submit** можно найти посредством команды `cy.contains`.

Рис. 8.7. Тест Cypress, исполняющийся в браузере

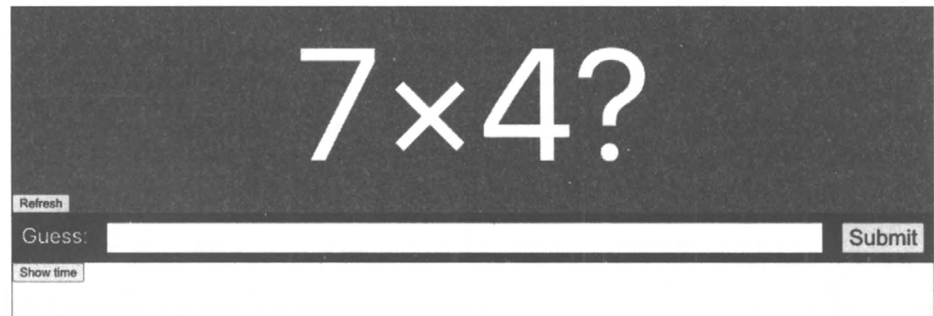


Рис. 8.8. Приложение дает указание пользователю вычислить произведение двух произвольных чисел

Листинг 8.15. Использование команд cy.get и cy.contains

```
describe('Basic game functions', () => {
  it('should notify the server if I lose', () => {
    // При условии, что приложение запущено
    cy.visit('http://localhost:3000')
    // Когда вводится неправильный ответ
    cy.get('input').type('101')
    cy.contains('Submit').click()
    // Сервер сообщает о проигрыше
  })
})
```

Теперь нам нужно только протестировать, что приложение обращается к серверу с результатом ответа.

Для этого мы используем команду `cy.intercept()`. Эта команда изменяет поведение сетевых запросов в приложении таким образом, чтобы можно было сымитировать ответ на данный запрос. Если результат будет отправляться методом `POST` в концевую точку /api/result, то сымитированный ответ создается, как показано в листинге 8.16.

Листинг 8.16. Создание сымитированного ответа

```
cy.intercept('POST', '/api/result', {
  statusCode: 200,
  body: '',
})
```

Когда эта команда возымеет действие, сетевые запросы к концевой точке /api/result будут получать сымитированный ответ. Это означает, что прежде чем отправлять сетевой запрос, нам нужно исполнить команду `cy.intercept()`. Мы будем делать это в начале теста (листинг 8.17).

Листинг 8.17. Исполнение команды cy.intercept

```
describe('Basic game functions', () => {
  it('should notify the server if I lose', () => {
    // При условии, что приложение запущено
    cy.intercept('POST', '/api/result', {
      statusCode: 200,
      body: '',
    })
    cy.visit('http://localhost:3000')
    // Когда вводится неправильный ответ
    cy.get('input').type('101')
    cy.contains('Submit').click()
```

```
      // Сервер сообщает о проигрыше
   })
})
```

Итак, мы подробно определили ответ сервера. Но как нам подтвердить, что приложение осуществило сетевой вызов, и как мы узнаем, что оно отправило правильные данные в концевую точку /api/result?

Нам нужно присвоить сетевому вызову псевдоним (листинг 8.18). Это также позволит нам обращаться к запросу далее в тесте[9].

Листинг 8.18. Присвоение псевдонима сетевому вызову

```
cy.intercept('POST', '/api/result', {
   statusCode: 200,
   body: '',
}).as('postResult')
```

Затем в конце теста мы можем создать утверждение, которое будет ожидать подачи сетевого вызова и проверять содержимое данных, отправляемых в теле запроса (листинг 8.19).

Листинг 8.19. Ожидание сетевого запроса и проверка содержимого данных

```
describe('Basic game functions', () => {
   it('should notify the server if I lose', () => {
      // При условии, что приложение запущено
      cy.intercept('POST', '/api/result', {
         statusCode: 200,
         body: '',
      }).as('postResult')
      cy.visit('http://localhost:3000')
      // Когда вводится неправильный ответ
      cy.get('input').type('101')
      cy.contains('Submit').click()
      // Сервер сообщает о проигрыше
      cy.wait('@postResult').then((xhr) => {
         expect(xhr.request.body.guess).equal(101)
         expect(xhr.request.body.result).equal('LOSE')
      })
   })
})
```

[9] Команда `cy.intercept` не может просто возвратить ссылку на сымитированный сетевой запрос вследствие помещения команд Cypress в цепочку.

Данное утверждение проверяет два атрибута тела запроса на наличие ожидаемых значений.

Если теперь исполнить наш тест, то он будет успешно выполнен, как показано на рис. 8.9.

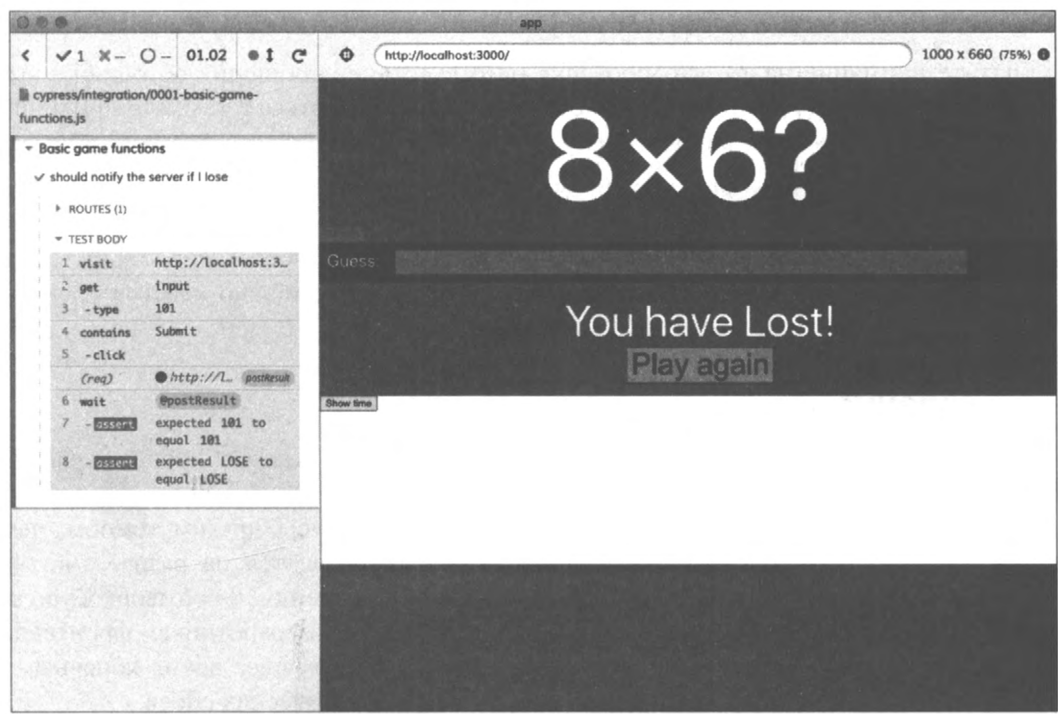


Рис. 8.9. Успешное завершение теста

Теперь, когда у нас есть тест для успешного случая, можно приступать к написанию теста для неуспешного случая. В частности, при сетевом сбое приложение должно отобразить на экране соответствующее сообщение. По сути, для нас не важны подробности, отправляемые на сервер в этом тесте, но нам все равно нужно ожидать завершения сетевого запроса, прежде чем выполнять проверку на наличие сообщения об ошибке. Код теста приведен в листинге 8.20.

Листинг 8.20. Код теста для сетевых сбоев

```
it('should display a message if I cannot post the result', () => {
  // При условии, что приложение запущено
  cy.intercept('POST', '/api/result', {
    statusCode: 500,
    body: { message: 'Bad thing happened!' },
  }).as('postResult')
  cy.visit('http://localhost:3000')
  // Когда вводится ответ
  cy.get('input').type('16')
```

```
  cy.contains('We are unable to save the result').should('not.exist')
  cy.contains('Submit').click()
  // Выводится сообщение об ошибке
  cy.wait('@postResult')
  cy.contains('We are unable to save the result')
})
```

Обратите внимание на то, что проверка на отсутствие сообщения об ошибке выполняется до осуществления сетевого вызова, чтобы убедиться в том, что причиной ошибки является именно сетевой вызов.

Кроме генерирования фиксированных кодов состояния и ответов команда `cy.intercept` может выполнять и другие операции, такие как замедление времени ответа, дросселирование скорости сети или генерирование ответов от тестовых функций. Дополнительную информацию по этой команде можно найти в ее документации, доступной по адресу **https://oreil.ly/tcZR8**.

Обсуждение

Тестирование при помощи фреймворка Cypress может изменить к лучшему работу команды разработки, в частности придав ей способность имитировать сетевые вызовы. Программисты часто разрабатывают API-интерфейсы другим темпом, чем код фронтенда. Кроме того, некоторые команды разделяются на разработчиков, которые специализируются или по фронтенду, или по бэкенду. Фреймворк Cypress может быть полезен в таких ситуациях, т. к. он позволяет разработчикам фронтенда создавать код для работы с еще несуществующими в настоящее время конечными точками. Он также может эмулировать все случаи патологических сбоев.

Производительность сети может вносить в работу приложения несистематические ошибки. В средах разработки используются локальные серверы с небольшими объемами или вообще отсутствующими данными. Это означает, что производительность API-интерфейса будет намного лучшей при разработке, чем в рабочей среде. Написать код, предполагающий немедленную доступность данных, очень легко, но в рабочей среде, где для прихода данных может потребоваться секунда или около этого, такой код окажется неработоспособным.

Поэтому целесообразно иметь, по крайней мере, один тест для каждого вызова API-интерфейса, где ответ прибывает с опозданием в секунду или около этого (листинг 8.21).

Листинг 8.21. Тест с задержкой доступности данных

```
cy.intercept('GET', '/api/widgets', {
  statusCode: 200,
  body: [{ id: 1, name: 'Flange' }],
  delay: 1000,
}).as('getWidgets')
```

Эмулирование медленных сетевых ответов часто выявляет целую кучу асинхронных ошибок, которые в противном случае могут прокрасться в код.

Очень важно, что создание искусственно замедленных сетевых ответов позволит вам ощутить общее воздействие на производительность каждого вызова API-интерфейса.

Исходный код данного рецепта можно загрузить на веб-сайте GitHub по адресу **https://oreil.ly/PITqj**.

8.4. Использование Cypress для офлайнового тестирования

ЗАДАЧА

В этом рецепте используется специализированная команда Cypress, разработанная Этьеном Бруинесом (Etienne Bruines) (**https://oreil.ly/oOMHP**).

Приложения должны быть способными справляться с потерей подключения к сети. Мы уже рассмотрели в *разделе 3.5*, как создать хук для определения состояния подключения к сети. Но как протестировать офлайновое поведение?

РЕШЕНИЕ

Работу в офлайне можно эмулировать при помощи фреймворка Cypress. Тесты Cypress могут внедрять код, который модифицирует внутреннее поведение тестируемого браузера. Поэтому мы должны быть в силах модифицировать сетевой код, чтобы эмулировать офлайновые условия.

Для этого рецепта в приложение необходимо установить фреймворк Cypress. Если фреймворк еще не установлен, установите его, выполнив следующую команду в папке приложения:

```
$ npm install --save-dev cypress
```

Затем в папке cypress/integration создайте файл 0002-offline-working.js и вставьте в него код из листинга 8.22.

Листинг 8.22. Содержимое файла 0002-offline-working.js

```
describe('Offline working', () => {
  it(
    'should tell us when we are offline',
    // Тест должен сообщить нам, когда мы в офлайне
    { browser: ''firefox' },
    () => {
      // При условии, что приложение запущено
      // Когда приложение в офлайне,
      // выводится соответствующее предупреждение
```

```
            // Когда приложение возвращается в онлайн,
            // предупреждение не выводится
        }
    )
})
```

Этот тест игнорируется в браузере Firefox. Код эмулирования режима офлайн полагается на протокол удаленной отладки Chrome DevTools, который в настоящее время не поддерживается браузером Firefox.

Структура теста обозначена последовательностью комментариев. Все команды Cypress начинаются с префикса `cy`, поэтому приложение можно запустить, как показано в листинге 8.23.

Листинг 8.23. Запуск приложения с Cypress

```
describe('Offline working', () => {
    it(
        'should tell us when we are offline',
         // Тест должен сообщить нам, когда мы в офлайне
        { browser: '!firefox' },
        () => {
            // При условии, что приложение запущено
            cy.visit('http://localhost:3000')
            // Когда приложение в офлайне,
            // выводится соответствующее предупреждение
            // Когда приложение возвращается в онлайн,
            // предупреждение не выводится
        }
    )
})
```

Вопрос в том, как нам заставить браузер эмулировать работу в офлайне?

Это можно сделать благодаря расширяемости фреймворка Cypress. Добавление специализированной команды Cypress `cy.network` позволит нам переключаться между офлайновым и онлайновым режимами:

```
cy.network({ offline: true })
cy.network({ offline: false })
```

Добавление этой команды осуществляется вставкой кода, приведенного в листинге 8.24, в файл cypress/support/command.js.

Листинг 8.24. Код для добавления команды cy.network

```
Cypress.Commands.add('network', (options = {}) => {
    Cypress.automation('remote:debugger:protocol', {
```

```
      command: 'Network.enable',
   })
   Cypress.automation('remote:debugger:protocol', {
      command: 'Network.emulateNetworkConditions',
      params: {
         offline: options.offline,
         latency: 0,
         downloadThroughput: 0,
         uploadThroughput: 0,
         connectionType: 'none',
      },
   })
})
```

Команда `cy.network` эмулирует офлайновые сетевые условия посредством протокола удаленной отладки DevTools. Сохранив этот файл, можно приступать к реализации остальной части теста (листинг 8.25).

Листинг 8.25. Полный код теста для работы в офлайне

```
describe('Offline working', () => {
   it(
      'should tell us when we are offline',
      // Тест должен сообщить нам, когда мы в офлайне
      { browser: '!firefox' },
      () => {
         // При условии, что приложение запущено
         cy.visit('http://localhost:3000')
         cy.contains(/you are currently offline/i).should('not.exist')
         // Когда приложение в офлайне
         cy.network({ offline: true })
         // Выводится соответствующее предупреждение
         cy.contains(/you are currently offline/i).should('be.visible')
         // Когда приложение возвращается в онлайн
         cy.network({ offline: false })
         // Предупреждение не выводится
         cy.contains(/you are currently offline/i).should('not.exist')
      }
   )
})
```

Если теперь исполнить наш тест, то он будет успешно выполнен, как показано на рис. 8.10.

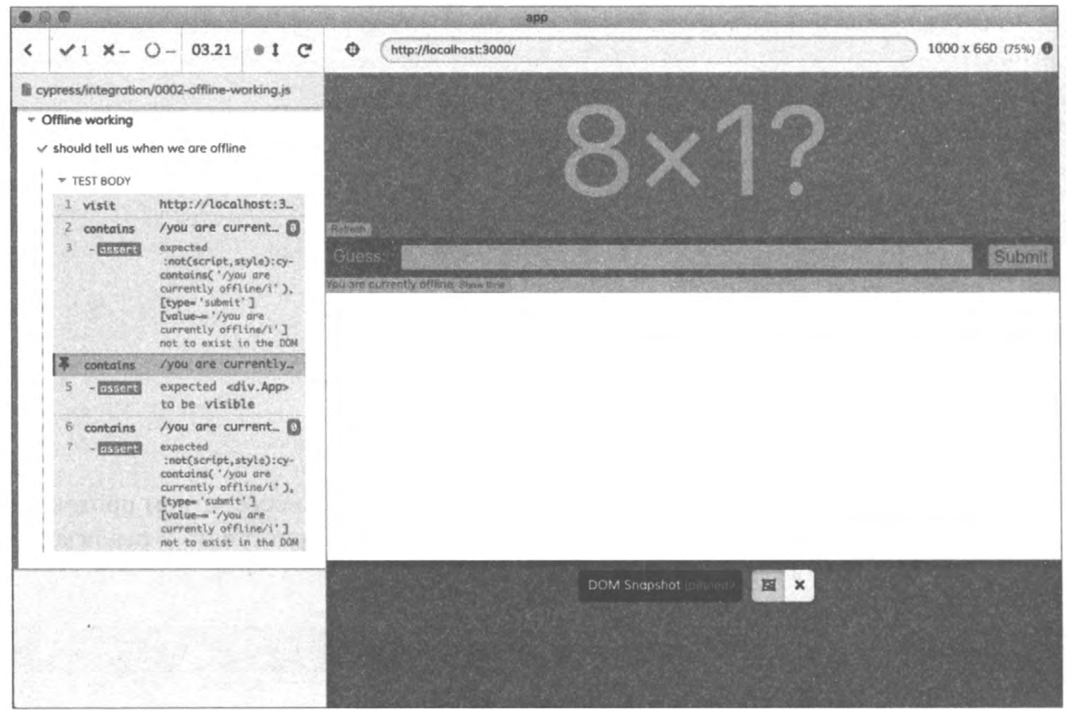


Рис. 8.10. Исполнение теста работы в онлайне и офлайне можно просматривать, щелкая по левой панели

Обсуждение

Подобные команды можно создать для эмулирования различных сетевых условий и скоростей.

Подробная информация о работе команды `cy.network` предоставляется в блоге компании Cypress.io по адресу **https://oreil.ly/PB4zO**.

Исходный код данного рецепта можно загрузить на веб-сайте GitHub по адресу **https://oreil.ly/PITqj**.

8.5. Использование Selenium для тестирования в браузере

ЗАДАЧА

Нет лучшего способа тестирования кода, чем его исполнение в настоящем браузере, а применение веб-драйвера — наиболее распространенный способ создания автоматизированных тестов для исполнения в браузере. Большинством браузеров можно управлять, посылая команду на сетевой порт. Для разных браузеров существуют разные команды, и инструмент командной строки в виде веб-драйвера упрощает управление браузером.

Но как можно создать тест для приложения React, который использует веб-драйвер?

РЕШЕНИЕ

Для решения этой задачи мы воспользуемся фреймворком Selenium, который предоставляет единообразный API-интерфейс для целого ряда разных веб-драйверов. Это означает, что код теста для браузера Firefox должен работать также с браузерами Chrome, Safari и Edge[10].

Для этого рецепта мы возьмем то же самое приложение, что и для остальных рецептов в этой главе. Это математическая игра, в которой пользователь должен решить простую задачу на умножение.

Версии библиотеки Selenium доступны для целого ряда разных языков программирования, например Python, Java и C#. Мы выбрали версию для JavaScript: Selenium WebDriver.

Начнем с установки этого фреймворка:

```
$ npm install --save-dev selenium-webdriver
```

Также нам нужно установить, по крайней мере, один веб-драйвер. Веб-драйверы можно устанавливать глобально, но работа с ними будет более управляемой при установке в разрабатываемое приложение. Мы установим веб-драйвер `chromedriver` для браузера Chrome (для браузера Firefox нужно установить драйвер `geckodriver`):

```
$ npm install --save-dev chromedriver
```

Теперь можно приступать к созданию теста. Целесообразно разместить тесты Selenium в папке src приложения, т. к. это облегчит использование среды разработки для ручного исполнения тестов. Далее, создадим в папке src подпапку selenium, а в ней файл 0001-basic-game-funcions.spec.js и вставим в этот файл код из листинга 8.26[11].

Листинг 8.26. Содержимое файла 0001-basic-game-functions.spec.js

```
describe('Basic game functions', () => {
   it('should tell me if I won', () => {
      // Тест должен сообщить нам о выигрыше
      // При условии, что приложение запущено
      // При вводе правильного ответа
      // Отображается сообщение о выигрыше
   })
})
```

[10] Это не означает, что тесты будут работать с каждым браузером, а гарантирует только то, что они будут исполняться на всех браузерах.

[11] Мы просто следуем соглашению, согласно которому название каждого теста начинается со связанного номера истории. Но для Selenium это не обязательно.

Содержимое теста изложено в общих чертах в комментариях.

Несмотря на то что размещение тестов Selenium в дереве папок src делает удобной работу с ними, это может вызывать их исполнение такими инструментами, как Jest, как будто бы они были модульными тестами. Может возникнуть проблема, если вы непрерывно исполняете модульные тесты в фоновом режиме. Например, если вы создали приложение с помощью средства `create-react-com` и оставили исполняющейся команду `npm run test`, то обнаружите, что при каждом сохранении теста Selenium на экране будет внезапно открываться окно браузера. Этого можно избежать, приняв какой-либо определенный способ именования файлов, чтобы различать между тестами Selenium и модульными тестами. Например, если все файлы тестов Selenium именовать в формате *.spec.js, то тестовый сценарий можно модифицировать, чтобы избегать их, указав в нем тесты *react-scripts '.*.test.js'*.

Фреймворк Selenium автоматизирует веб-браузер при помощи веб-драйвера. Экземпляр веб-драйвера можно создавать в начале каждого теста, как показано в листинге 8.27.

Листинг 8.27. Создание экземпляра веб-драйвера

```
import { Builder } from 'selenium-webdriver'
let driver
describe('Basic game functions', () => {
   beforeEach(() => {
      driver = new Builder().forBrowser('chrome').build()
   })
   afterEach(() => {
      driver.quit()
   })
   it('should tell me if I won', () => {
      // Тест должен сообщить нам о выигрыше
      // При условии, что приложение запущено
      // При вводе правильного ответа
      // Отображается сообщение о выигрыше
   })
})
```

В данном примере создается веб-драйвер для браузера Chrome.

Создавая веб-драйвер для каждого теста, мы также создаем новый экземпляр браузера для каждого теста, тем самым гарантируя, что между тестами нет никаких промежуточных состояний браузера. Это позволит нам исполнять тесты в любом порядке, что было бы невозможно при наличии общих состояний браузера. Если тесты полагаются, например, на информацию из базы данных, необходимо обеспечить правильную инициализацию сервера тестом при его запуске.

Кроме того, чтобы Selenium мог создать экземпляр драйвера, нам нужно явно затребовать драйвер, как показано в листинге 8.28.

Листинг 8.28. Вызов драйвера

```
import { Builder } from 'selenium-webdriver'
require('chromedriver')
let driver
describe('Basic game functions', () => {
  beforeEach(() => {
    driver = new Builder().forBrowser('chrome').build()
  })
  afterEach(() => {
    driver.quit()
  })
  it('should tell me if I won', () => {
    // Должно появиться сообщение о выигрыше
    // При условии, что приложение запущено
    // При вводе правильного ответа
    // Отображается сообщение о выигрыше
  })
})
```

Теперь можно приступать к созданию остального кода теста. Версия Selenium для JavaScript имеет высокую степень асинхронности. Практически все команды возвращают обещания, что означает высокую эффективность, но также создает очень благоприятные условия для возникновения ошибок тестирования.

Начнем наш тест с открытия приложения, как показано в листинге 8.29.

Листинг 8.29. Открытие приложения

```
import { Builder } from 'selenium-webdriver'
require('chromedriver')
let driver
describe('Basic game functions', async () => {
  beforeEach(() => {
    driver = new Builder().forBrowser('chrome').build()
  })
  afterEach(() => {
    driver.quit()
  })
  it('should tell me if I won', () => {
    // Тест должен сообщить нам о выигрыше
    // При условии, что приложение запущено
    await driver.get('http://localhost:3000')
    // При вводе правильного ответа
    // Отображается сообщение о выигрыше
  }, 60000)
})
```

Команда `driver.get` дает указание браузеру открыть данный URL-адрес. Но чтобы эта команда могла работать, нам нужно было внести еще две модификации. Первое, мы пометили функцию тестирования ключевым словом `async`, что позволит нам ожидать обещание, возвращаемое методом `driver.get`.

Второе, мы добавили в тест значение тайм-аута величиной в 60 000 миллисекунд, заменив неявный пятисекундный предел тестов Jest. Если не увеличить значение тайм-аута, то тест завершится неуспешно прежде, чем запустится браузер. В данном случае значение 60 000 миллисекунд было выбрано, чтобы обеспечить работу теста на любой машине. Это значение можно откорректировать в соответствии с имеющимся оборудованием.

Чтобы ввести правильный ответ на вопрос, нам нужно прочитать два числа этого вопроса (рис. 8.11).

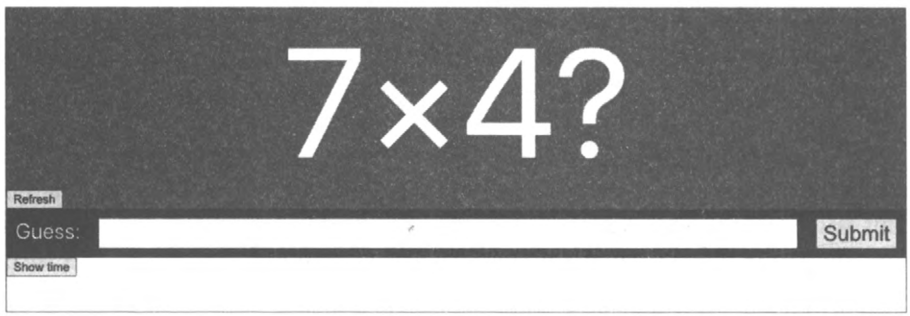


Рис. 8.11. В игре нужно вычислить произведение двух произвольных чисел

Эти два числа, а также кнопки `input` и `submit` на странице можно найти при помощи команды `findElement`, как показано в листинге 8.30.

Листинг 8.30. Использование команды `findElement`

```
const number1 = await driver.findElement(By.css('.number1')).getText()
const number2 = await driver.findElement(By.css('.number2')).getText()
const input = await driver.findElement(By.css('input'))
const submit = await driver.findElement(
   By.xpath("//button[text()='Submit']")
)
```

Если при считывании со страницы набора элементов точный порядок их обработки не имеет значения, то при помощи функции `Promise.all` все эти элементы можно объединить в одно обещание, которое потом можно ожидать (листинг 8.31).

Листинг 8.31. Объединение нескольких элементов в одно обещание

```
const [number1, number2, input, submit] = await Promise.all([
   driver.findElement(By.css('.number1')).getText(),
```

```
  driver.findElement(By.css('.number2')).getText(),
  driver.findElement(By.css('input')),
  driver.findElement(By.xpath("//button[text()='Submit']")),
])
```

В нашем примере приложения данная оптимизация практически не экономит нам время, но если страница отрисовывает разные компоненты в неопределенном порядке, то объединение обещаний может улучшить производительность теста.

Теперь мы можем реализовать следующую часть нашего теста (листинг 8.32).

Листинг 8.32. Использование команды `findElement` в тесте

```
import { Builder, By } from 'selenium-webdriver'
require('chromedriver')
let driver
describe('Basic game functions', async () => {
  beforeEach(() => {
    driver = new Builder().forBrowser('chrome').build()
  })
  afterEach(() => {
    driver.quit()
  })
  it('should tell me if I won', () => {
    // Тест должен сообщить нам о выигрыше
    // При условии, что приложение запущено
    await driver.get('http://localhost:3000')
    // При вводе правильного ответа
    const [number1, number2, input, submit] = await Promise.all([
      driver.findElement(By.css('.number1')).getText(),
      driver.findElement(By.css('.number2')).getText(),
      driver.findElement(By.css('input')),
      driver.findElement(By.xpath("//button[text()='Submit']")),
    ])
    await input.sendKeys('' + number1 * number2)
    await submit.click()
    // Отображается сообщение о выигрыше
  }, 60000)
})
```

Обратите внимание на то, что мы объединяем обещания, возвращаемые методами `input.sendKeys` и `submit.click`, поскольку хотим, чтобы тест ввел ответ в поле ввода *прежде*, чем мы его отправим.

Наконец, мы хотим выполнить утверждение, что сообщение "You have won!" будет отображено на экране (рис. 8.12).

Это утверждение можно написать, как показано в листинге 8.33.

Рис. 8.12. Приложение сообщает пользователю о вводе им правильного ответа

Листинг 8.33. Один из вариантов написания утверждения о выигрыше

```
const resultText = await driver
   .findElement(By.css('.Result'))
   .getText()
expect(resultText).toMatch(/won/i)
```

Этот код почти наверняка будет работоспособным, поскольку результат отображается почти сразу после предоставления ответа пользователем. Приложения React часто медлят с отображением динамических результатов, особенно если они полагаются на сетевые данные. Но если модифицировать код приложения, чтобы эмулировать двухсекундную задержку перед отображением результата[12], то наш тест выдаст сообщение об ошибке, приведенное в листинге 8.34.

Листинг 8.34. Сообщение об ошибке при двухсекундной задержке отображения результата

```
no such element: Unable to locate element: {"method":"css selector",
      "selector":".Result"}
   (Session info: chrome=88.0.4324.192)
NoSuchElementError: no such element: Unable to locate element: {
      "method":"css selector","selector":".Result"}
   (Session info: chrome=88.0.4324.192)
```

Этой проблемы можно избежать, ожидая до тех пор, пока элемент не появится на экране, а затем подождав еще некоторое время, пока текст не совпадет с ожидае-

[12] Соответствующий код находится в исходном коде для этой главы, который можно загрузить с репозитория GitHub по адресу **https://oreil.ly/PITqj**.

мым результатом. Оба интервала ожидания можно реализовать при помощи метода `driver.wait`, как показано в следующем фрагменте кода:

```
await driver.wait(until.elementLocated(By.css('.Result')))
const resultElement = driver.findElement(By.css('.Result'))
await driver.wait(until.elementTextMatches(resultElement, /won/i))
```

Это позволяет нам создать окончательную версию нашего теста, приведенную в листинге 8.35.

Листинг 8.35. Окончательная версия теста

```
import { Builder, By } from 'selenium-webdriver'
require('chromedriver')
let driver
describe('Basic game functions', async () => {
  beforeEach(() => {
    driver = new Builder().forBrowser('chrome').build()
  })
  afterEach(() => {
    driver.quit()
  })
  it('should tell me if I won', () => {
    // Тест должен сообщить нам о выигрыше
    // При условии, что приложение запущено
    await driver.get('http://localhost:3000')
    // При вводе правильного ответа
    const [number1, number2, input, submit] = await Promise.all([
      driver.findElement(By.css('.number1')).getText(),
      driver.findElement(By.css('.number2')).getText(),
      driver.findElement(By.css('input')),
      driver.findElement(By.xpath("//button[text()='Submit']")),
    ])
    await input.sendKeys('' + number1 * number2)
    await submit.click()
    // Отображается сообщение о выигрыше
    await driver.wait(until.elementLocated(By.css('.Result')))
    const resultElement = driver.findElement(By.css('.Result'))
    await driver.wait(until.elementTextMatches(resultElement, /won/i))
  }, 60000)
})
```

Обсуждение

На основе своего опыта мы можем сказать, что веб-драйверы являются наиболее часто используемой формой автоматизированных тестов для веб-приложений. Эти тесты неизбежно зависят от наличия совпадающих версий браузера и веб-драйвера, а также имеют репутацию склонных к спорадическим сбоям. Такие ошибки обычно

вызваны проблемами с таймингом, которые чаще всего происходят в одностраничных приложениях, способных к асинхронному обновлению своего содержимого.

Указанных проблем можно избежать, тщательно добавляя в код временные задержки и повторные попытки, но это может сделать тесты чувствительными к изменениям в среде исполнения, например при исполнении приложения на другом сервере для тестирования. В качестве альтернативного решения этой проблемы, в случае большого количества спорадических сбоев, можно переместить бóльшую часть тестов на систему наподобие Cypress, которая менее чувствительна к ошибкам с таймингом (**https://oreil.ly/IZJ2T**).

Исходный код данного рецепта можно загрузить на веб-сайте GitHub по адресу **https://oreil.ly/PITqj**.

8.6. Тестирование внешнего вида на разных браузерах посредством ImageMagick

ЗАДАЧА

Внешний вид приложений может существенно отличаться при просмотре в разных браузерах. Более того, они могут выглядеть по-другому даже на том же самом браузере, но на другой операционной системе. Например, браузер Chrome обычно скрывает полосы прокрутки при исполнении на компьютерах Mac, но отображает их на компьютерах Windows. К счастью, старые браузеры наподобие Internet Explorer наконец-то начинают исчезать, но даже современные браузеры могут применять CSS слегка по-другому, коренным образом изменяя внешний вид страницы.

Но постоянная проверка внешнего вида приложения вручную на ряде браузеров и платформ может оказаться довольно трудоемкой задачей.

Можно ли каким-либо образом автоматизировать этот процесс проверки совместимости?

РЕШЕНИЕ

Да, можно. И в этом рецепте мы продемонстрируем, как это делается совместными усилиями трех инструментов:

- **Storybook** — предоставляет нам базовую галерею всех компонентов во всех релевантных конфигурациях, которые нужно проверить.
- **Selenium** — позволяет выполнять захват внешнего вида всех компонентов Storybook. А прокси-сервер Selenium Grid даст возможность выполнять удаленное подключение к браузерам, функционирующим на разных операционных системах, чтобы сравнить внешний вид приложений в каждом случае.
- **ImageMagick** — в частности, мы воспользуемся инструментом `compare` этого комплекта программ для выявления различий между двумя снимками экрана и предоставления числового значения величины этих различий.

Начнем с установки среды разработки Storybook, исполнив в папке приложения следующую команду:

```
$ npx sb init
```

Далее нам необходимо создать рассказы для каждого из компонентов и конфигураций, которые нужно отслеживать. Информацию о том, как это делается, можно найти в других рецептах этой книги или в учебных пособиях по Storybook (**https://oreil.ly/ak7VW**).

Затем нам нужно установить фреймворк Selenium, чтобы автоматизировать процесс захвата снимков экрана. Эта установка выполняется следующей командой:

```
$ npm install --save-dev selenium-webdriver
```

Также потребуется установить соответствующие веб-драйверы для тестируемых браузеров. Например, чтобы автоматизировать браузеры Firefox и Chrome, нужно установить следующие веб-драйверы:

```
$ npm install --save-dev geckodriver
$ npm install --save-dev chromedriver
```

Наконец, необходимо установить пакет инструментов командной строки для манипулирования изображениями ImageMagick. Информация по установке этого пакета доступна на странице для его загрузки (**https://oreil.ly/NIQ0A**).

Для данного примера вновь воспользуемся приложением математической игры, с которым мы работали ранее в этой главе. На рис. 8.13 показано отображение компонентов этого приложения в Storybook.

Сервер Storybook для приложения запускается следующей командой:

```
$ npm run storybook
```

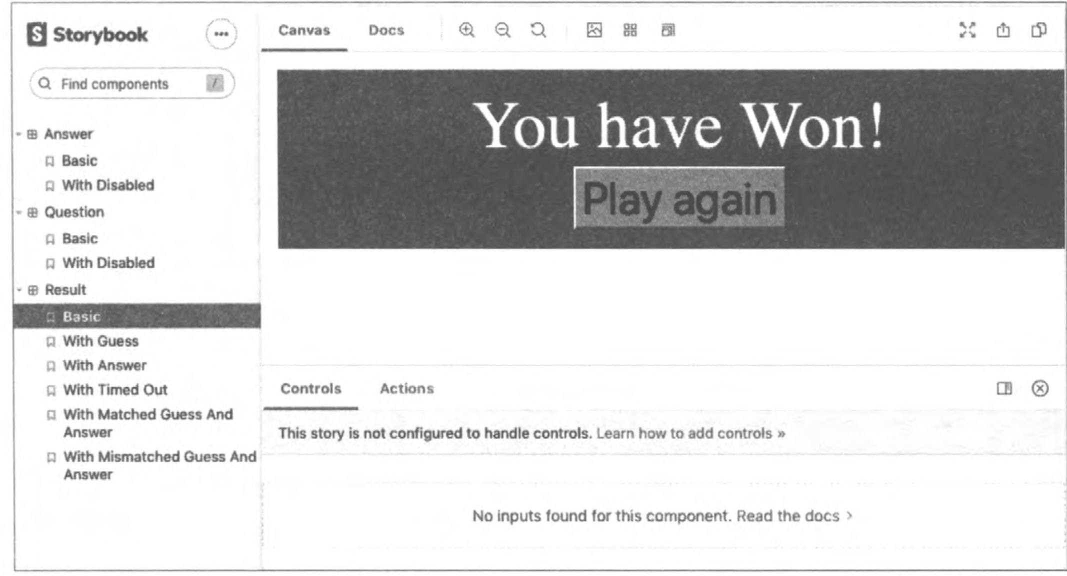


Рис. 8.13. Отображение компонентов приложения в Storybook

Теперь создадим тест в виде простого сценария для захвата снимков экрана каждого из компонентов в Storybook. Для этого в папке src/selenium создайте файл shots.spec.js и вставьте в него код из листинга 8.36[13].

Листинг 8.36. Код сценария shots.spec.js

```
import { Builder, By, until } from 'selenium-webdriver'
require('chromedriver')
let fs = require('fs')
describe('shots', () => {
   it('should take screenshots of storybook components', async () => {
  // Тест должен сделать снимки экрана компонентов Storybook
      const browserEnv = process.env.SELENIUM_BROWSER || 'chrome'
      const url = process.env.START_URL || 'http://localhost:6006'
      const driver = new Builder().forBrowser('chrome').build()
      driver.manage().window().setRect({
         width: 1200,
         height: 900,
         x: 0,
         y: 0,
      })
      const outputDir = './screenshots/' + browserEnv
      fs.mkdirSync(outputDir, { recursive: true })
      await driver.get(url)
      await driver.wait(
         until.elementLocated(By.className('sidebar-item')),
            60000
      )
      let elements = await driver.findElements(
         By.css('button.sidebar-item')
      )
      for (let e of elements) {
         const expanded = await e.getAttribute('aria-expanded')
         if (expanded !== 'true') {
            await e.click()
         }
      }
      let links = await driver.findElements(By.css('a.sidebar-item'))
      for (let link of links) {
         await link.click()
         const s = await link.getAttribute('id')
         let encodedString = await driver
            .findElement(By.css('#storybook-preview-wrapper'))
            .takeScreenshot()
```

[13] В принципе, этот сценарий можно было бы сохранить в любой папке, но мы использовали данную папку в коде примера на веб-сайте GitHub.

```
        await fs.writeFileSync(
          `${outputDir}/${s}.png`,
          encodedString,
          'base64'
        )
      }
      driver.quit()
    }, 60000)
})
```

Данный сценарий запускает браузер для сервера Storybook, открывает каждый из компонентов и делает снимок экрана каждой истории, которые он сохраняет в подпапке в screenshots.

Для захвата снимков экрана компонентов можно было бы использовать другую систему тестирования, например Cypress. Но преимущество Selenium состоит в том, что мы можем удаленно открыть браузерную сессию на компьютере.

По умолчанию тест shots.spec.js будет делать снимки экрана компонентов Storybook по адресу **http://localhost:6006** в браузере Chrome. Предположим, что это приложение тестирования исполняется на компьютере Mac. Если у нас в этой же сети есть компьютер Windows, мы можем установить прокси-сервер Selenium Grid (**https://oreil.ly/gYLds**), который позволит удаленным компьютерам запускать сессию веб-драйвера.

Если компьютер Windows имеет адрес `192.168.1.16`, то, прежде чем запускать на исполнение тест shots.spec.js, мы можем присвоить значение соответствующей переменной среды из командной строки:

```
$ export SELENIUM_REMOTE_URL=http://192.168.1.16:4444/wd/hub
```

Поскольку компьютер Windows будет обращаться к серверу Storybook на компьютере Mac с IP-адресом, например, `192.168.1.14`, нам также нужно присвоить значение переменной среды и для этого компьютера:

```
$ export START_URL=http://192.168.1.14:6006
```

Также можно указать, какой браузер использовать на компьютере Windows[14]:

```
$ export SELENIUM_BROWSER=firefox
```

Далее в файле package.json создадим сценарий для запуска теста shots.spec.js (листинг 8.37).

Листинг 8.37. Сценарий для запуска теста shots.spec.js

```
"scripts": {
  ...
```

[14] Чтобы все это работало, на удаленном компьютере должны быть установлены соответствующий браузер и веб-драйвер.

```
  "testShots": "CI=true react-scripts test --detectOpenHandles \
                'selenium/shots.spec.js'"
}
```

Теперь мы можем запустить тест на исполнение и делать снимки экрана каждого компонента:

```
$ npm run testShots
```

Используя созданные нами ранее переменные среды, тест установит контакт с прокси-сервером Selenium Grid на удаленном компьютере. В частности, тест даст запрос прокси-серверу открыть браузер Firefox с подключением к локальному серверу Storybook. Затем тест отправит по сети на удаленный компьютер снимки экрана каждого компонента Storybook, сохраняя их в папке screenshots/firefox.

После исполнения теста для браузера Firefox исполняем его для браузера Chrome:

```
$ export SELENIUM_BROWSER=chrome
$ npm run testShots
```

Снимки экрана этого теста сохраняются в папке screenshots/chrome.

Более полная реализация этого метода может также сохранять данные об используемой операционной системе и типе клиента (например, размере экрана).

Теперь нужно проверить визуальные различия между снимками экрана для Chrome и Firefox, для чего нам пригодится пакет ImageMagick. Инструмент `compare` этого пакета может генерировать изображение, в котором выделяются визуальные различия между двумя сравниваемыми изображениями. Для примера рассмотрим два снимка экрана одного и того компонента с Firefox и Chrome, показанные на рис. 8.14.

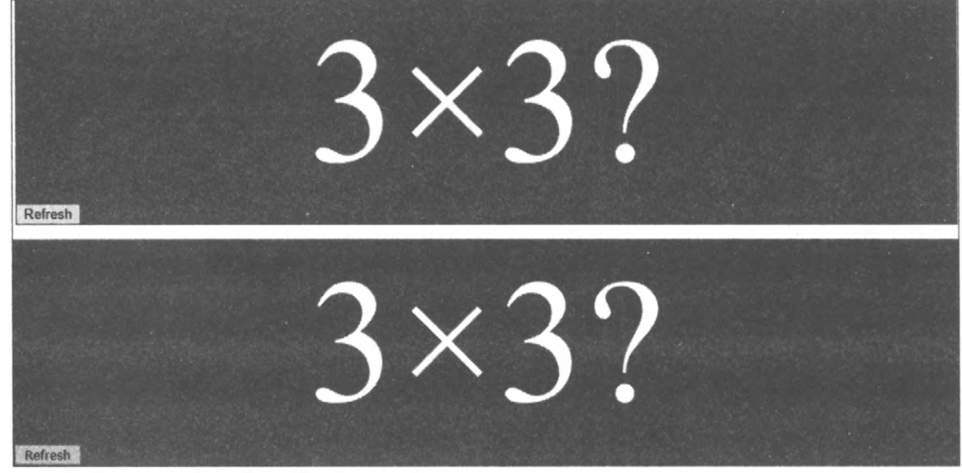


Рис. 8.14. Снимки экрана одного и того же компонента в Chrome и Firefox

На первый взгляд эти два изображения выглядят идентичными. В папке приложения выполним следующую команду:

```
$ compare -fuzz 15% screenshots/firefox/question--basic.png \
   screenshots/chrome/question--basic.png difference.png
```

Эта команда создает новое изображение (рис. 8.15), показывающее различия между двумя исходными изображениями.

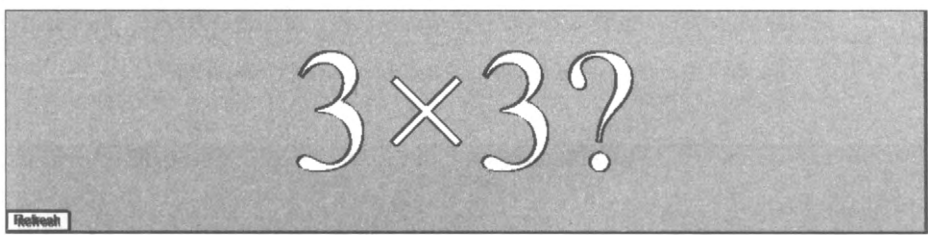


Рис. 8.15. Изображение, показывающее различия между двумя исходными изображениями снимков экрана

Созданное командой `compare` изображение состоит из пикселов двух исходных изображений, которые визуально различаются больше чем на 15%. При этом на глаз эти изображения снимков экрана кажутся практически идентичными.

Такая информация полезна, но все равно требует исследования изображений человеком, чтобы установить, насколько значительны различия. Какие еще сравнения можно выполнить?

Инструмент `compare` также может выдавать число, соответствующее различию между двумя исходными изображениями:

```
$ compare -metric AE -fuzz 15% screenshots/firefox/question--basic.png
   screenshots/chrome/question--basic.png difference.png
6774
```

Цифра `6774` представляет числовое значение (на основании подсчета абсолютных ошибок) визуальной разницы между двумя исходными изображениями. Для примера сравним два снимка экрана компонента `Answer` с включенным свойством `disabled` (рис. 8.16).

Для этих двух изображений команда `compare` возвращает намного большее числовое значение:

```
$ compare -metric AE -fuzz 15% screenshots/firefox/answer--with-disabled.png
   screenshots/chrome/answer--with-disabled.png difference3.png
28713
```

Рис. 8.16. Компонент Answer с включенным свойством disabled, отображенный в Chrome и Firefox

А на изображении визуальных различий этих двух снимков экрана (рис. 8.17) можно точно увидеть, где находятся эти различия: в отключенном поле ввода.

На рис. 8.18 приводится аналогичное значительное числовое значение различия (21 131) для компонента, шрифты которого отображаются по-разному в разных браузерах в результате некоторых CSS атрибутов, специфичных для браузера Mozilla.

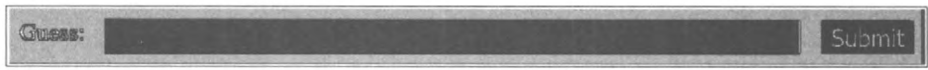


Рис. 8.17. Визуальные различия между формами Chrome и Firefox

Рис. 8.18. Текст компонента отображается по-разному в Chrome и Firefox

Собственно говоря, можно создать сценарий оболочки для обработки каждого изображения и формирования небольшого отчета, содержащего визуальные различия между изображениями вместе с числовыми значениями этих различий. Такой сценарий приведен в листинге 8.38.

Листинг 8.38. Сценарий для создания отчета о различиях в изображениях

```
#!/bin/bash
mkdir -p screenshots/diff
export HTML=screenshots/compare.html
echo '<body><ul>' > $HTML
for file in screenshots/chrome/*.png
do
   FROM=$file
   TO=$(echo $file | sed 's/chrome/firefox/')
   DIFF=$(echo $file | sed 's/chrome/diff/')
   echo "FROM $FROM TO $TO"
```

```
    ls -l $FROM
    ls -l $TO
    METRIC=$(compare -metric AE -fuzz 15% $FROM $TO $DIFF 2>&1)
    echo "<li>$FROM $METRIC<br/><img src=../$DIFF/></li>" >> $HTML
done
echo "</li></body>" >> $HTML
```

Создаваемый этим сценарием отчет (рис. 8.19) сохраняется в файле screenshots/compare.html.

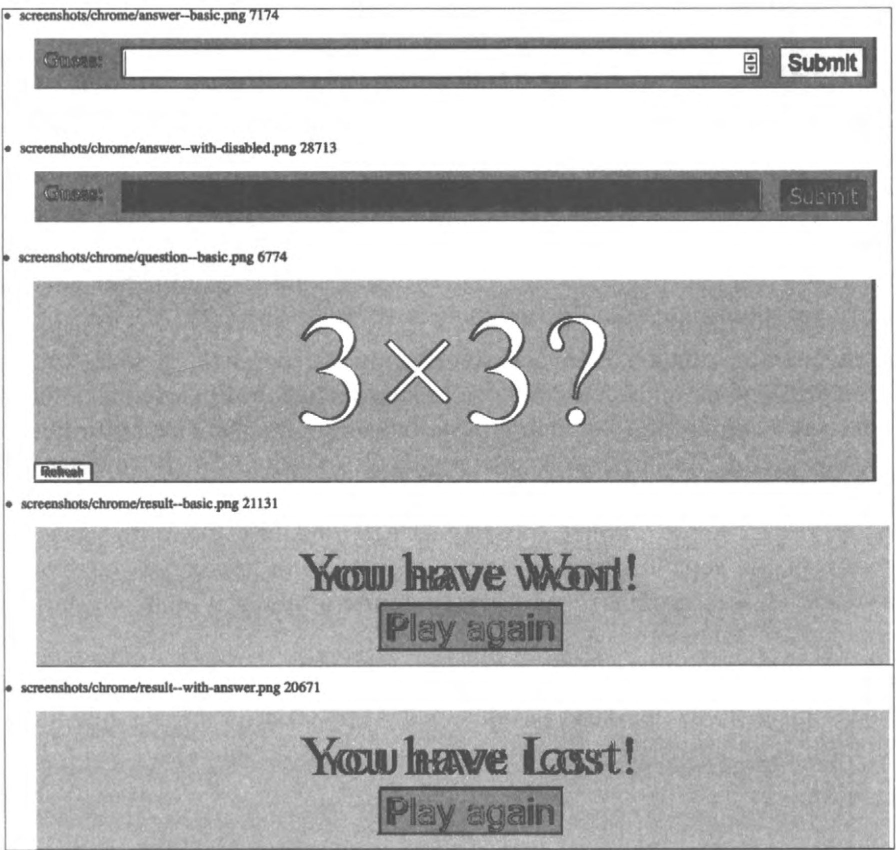


Рис. 8.19. Доклад о сравнении нескольких изображений

Обсуждение

Чтобы сэкономить место, мы показали только упрощенную реализацию этого метода. Но также можно создать ранжированный отчет, отображающий визуальные различия между парами изображений в порядке от наибольших до наименьших. Такой отчет покажет наиболее важные визуальные отличия между разными платформами.

Можно также использовать автоматизированные визуальные тесты, чтобы предотвратить регрессии. Здесь нужно не допускать ошибочных выявлений различий, вызываемых незначительными вариациями, такими как устранение контурных неровностей. С помощью непрерывной операции интегрирования можно задавать пороговое значение визуального различия между изображениями и генерировать сбой в случае, если величина различия между компонентами превышает величину этого порогового значения.

Исходный код данного рецепта можно загрузить на веб-сайте GitHub по адресу **https://oreil.ly/P1Tqj**.

8.7. Добавление консоли в браузер мобильного устройства

ЗАДАЧА

Приведенный далее рецепт немного отличается от других рецептов в этой главе тем, что в нем рассматривается не автоматизированное тестирование, а ручное, в частности тестирование кода на мобильных устройствах.

При тестировании приложения на мобильном устройстве можно столкнуться с ошибкой, которая не проявляется в среде настольных компьютеров. Обычно, при возникновении ошибки при исполнении приложения на персональном компьютере в консоли JavaScript браузера можно отображать отладочные сообщения. Но браузеры мобильных устройств, как правило, не оснащены консолью JavaScript. Хотя в случае браузера Mobile Chrome отладку на нем можно выполнять удаленно с помощью настольной версии Chrome. Но что делать, если проблема возникнет в другом браузере? Или если просто не хочется тратить время и силы на организацию сессии удаленной отладки?

Нет ли какого-либо способа доступа к консоли JavaScript, а также к другим инструментам разработки из браузера мобильного устройства?

РЕШЕНИЕ

Данную задачу можно решить при помощи программы Eruda (**https://oreil.ly/jCFSn**).

Это облегченная версия реализации панели инструментов разработки, которая позволяет просматривать консоль JavaScript, структуру страницы, а также целую кучу других модулей и расширений (**https://oreil.ly/ZUQHw**).

Чтобы разрешить работу с Eruda, нужно установить небольшой объем достаточно простого кода JavaScript в разделе `head` приложения. Загрузить Eruda на устройство можно из сети распространения контента. Однако размер этого инструмента довольно велик, поэтому его следует подключать только в том случае, если пользователь мобильного браузера указал, что хочет использовать его.

Один из способов, как это сделать, подключать Eruda только тогда, когда URL-адрес содержит строку `eruda=true`. В листинге 8.39 приведен код сценария, который можно вставить в контейнер страницы[15].

Листинг 8.39. Код сценария для подключения Eruda

```
<script>
  (function () {
    var src = '//cdn.jsdelivr.net/npm/eruda';
    if (!/eruda=true/.test(window.location)
      && localStorage.getItem('active-eruda') != 'true') return;
    document.write('<scr' + 'ipt src="' + src
      + '"></scr' + 'ipt>');
    document.write('<scr' + 'ipt>');
    document.write('window.addEventListener(' +
      '"load", ' +
      'function () {' +
      ' var container=document.createElement("div"); ' +
      ' document.body.appendChild(container);' +
      ' eruda.init({' +
      ' container: container,' +
      ' tool: ["console", "elements"]' +
      ' });' +
      '})');
    document.write('</scr' + 'ipt>');
  })();
</script>
```

Если теперь открыть приложение в браузере по адресу **http://ipaddress/?eruda=true** или **http://ipaddress/ #eruda=true**, то в его интерфейсе будет дополнительная кнопка, как показано на рис. 8.20.

Если вы используете пример приложения для этой главы, попробуйте ввести несколько ответов[16], а затем нажмите кнопку **Eruda**. В результате откроется консоль, как показано на рис. 8.21.

Поскольку конечная точка вызовов приложения отсутствует, то в консоли должны быть записаны некоторые ошибки и другие журналы. Консоль даже поддерживает очень малораспространенную функцию `console.table`, которая позволяет отображать массивы объектов в табличном формате.

Вкладка **Elements** панели **Eruda** предоставляет базовое представление модели DOM (рис. 8.22).

---

[15] Для приложений, созданных при помощи средства `create-react-app`, этот код нужно вставить в файл public/index.html.

[16] Код этого приложения можно загрузить с репозитория исходного кода для этой книги по адресу https://oreil.ly/P1Tqj.

Рис. 8.20. Дополнительная кнопка на странице (справа внизу) приложения

Рис. 8.21. Консоль инструментов **Eruda**

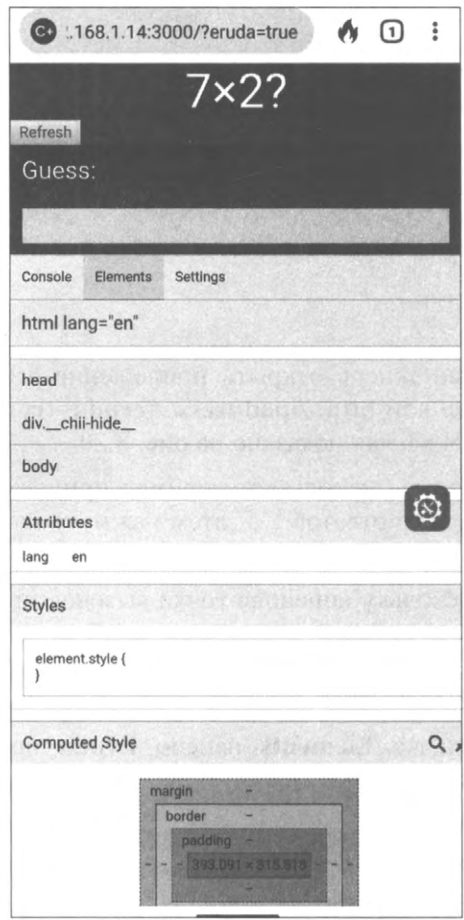


Рис. 8.22. Вкладка **Elements** панели **Eruda**

А вкладка **Settings** (рис. 8.23) предоставляет доступ к обширному набору возможностей JavaScript, которые можно включать и отключать в процессе взаимодействия с веб-страницей.

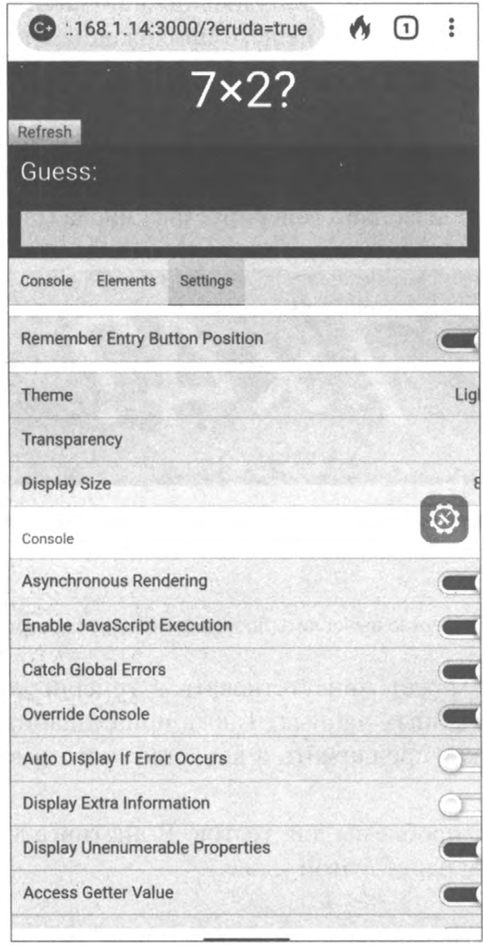


Рис. 8.23. Вкладка **Settings** панели **Eruda**

Обсуждение

Eruda — это чудесный инструмент, предоставляющий целый ряд возможностей, требующий очень небольшого объема работы со стороны разработчика. Кроме базовых возможностей он также поддерживает подключаемые модули, позволяющие отслеживать производительность, задавать скорость обновления экрана, генерировать ложные данные геолокации и даже исполнять JavaScript из браузера. Начав работать с ним, вы, скорее всего, обнаружите, что он быстро станет стандартной составляющей вашего процесса ручного тестирования.

Исходный код данного рецепта можно загрузить на веб-сайте GitHub по адресу **https://oreil.ly/P1Tqj**.

8.8. Удаление произвольности из тестов

ЗАДАЧА

В идеальном мире тесты всегда исполнялись бы в полностью искусственной среде. Тесты являются примерами работы нашего приложения в явно определенных условиях. Но тестам часто приходится иметь дело с неопределенными факторами. Например, они могут исполняться в разное время дня. В примере приложения, которое мы рассматриваем в этой главе, тоже есть случайные факторы.

Это приложение представляет собой математическую игру, в которой пользователь должен умножить два произвольно генерируемых числа (рис. 8.24).

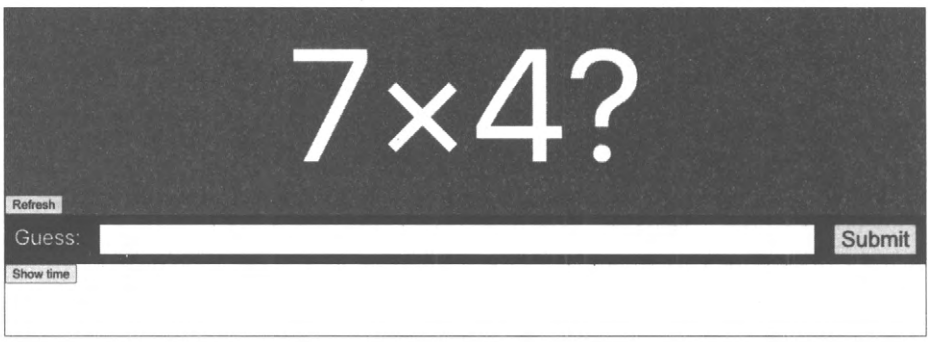


Рис. 8.24. В игре нужно вычислить произведение двух произвольных чисел

Произвольность также может присутствовать в генерировании идентификаторов в коде или наборов случайных данных. Также при создании нового имени пользователя приложение может предложить в качестве этого имени произвольно сгенерированную строку.

Но случайность создает проблемы для тестов. В листинге 8.40 приведен код теста, который мы реализовали ранее в этой главе.

Листинг 8.40. Код одного из предыдущих тестов главы

```
describe('Basic game functions', () => {
   it('should notify the server if I lose', () => {
      // Сообщает серверу о проигрыше
      // При условии, что приложение запущено
      // Когда вводится неправильный ответ
      // Серверу сообщается о проигрыше
   })
})
```

Неспроста этот тест был разработан для случая ввода пользователем неверного ответа. Для решения задачи всегда нужно вычислить произведение двух чисел от 1 до 10. Поэтому придумать и ввести неправильный ответ совсем не сложно: 101. Этот

ответ всегда будет неправильным. Но если мы хотим разработать тест, чтобы показать, что происходит, когда пользователь вводит правильный ответ, то сталкиваемся с проблемой. Правильный ответ зависит от произвольно генерируемых данных. Мы могли бы считывать выводимые на экран произвольные числа, написав для этого какой-либо код, как, например, в листинге 8.41.

Листинг 8.41. Код для считывания произвольных чисел с экрана

```
const [number1, number2, input, submit] = await Promise.all([
   driver.findElement(By.css('.number1')).getText(),
   driver.findElement(By.css('.number2')).getText(),
   driver.findElement(By.css('input')),
   driver.findElement(By.xpath("//button[text()='Submit']")),
])
await input.sendKeys('' + number1 * number2)
await submit.click()
```

Но иногда подобный подход невозможен. Например, фреймворк Cypress не обеспечивает прямой захват текстового содержимого со страницы. Поэтому разработка теста Cypress на ввод правильного ответа на поставленную задачу умножения будет сопряжена с большими трудностями. Это объясняется тем, что Cypress не позволяет выполнять захват данных со страницы и передавать их другим составляющим теста.

Намного лучше, если на время теста можно было бы отключить произвольность. Но возможно ли это?

РЕШЕНИЕ

Данную задачу можно решить, используя библиотеку Sinon, чтобы временно заменить метод `Math.random` созданной нами имитацией.

Сначала рассмотрим, как это можно сделать в модульном тесте. Создадим новый тест для высокоуровневого компонента `App`, который будет проверять, что ввод правильного значения вызывает отображение сообщения, сообщающего о выигрыше.

В тесте мы создадим функцию, которая будет заменять значение, возвращаемое методом `Math.random` каким-либо фиксированным значением (листинг 8.42).

Листинг 8.42. Функция для замены случайного значения постоянным

```
const sinon = require('sinon')
function makeRandomAlways(result) {
   if (Math.random.restore) {
      Math.random.restore()
   }
   sinon.stub(Math, 'random').returns(result)
}
```

Эта функция заменяет метод `random` объекта `Math` методом-заглушкой, который всегда возвращает одно и то же значение. Теперь эту функцию можно использовать в нашем тесте. Компонент `Question`, отображаемый на странице, всегда генерирует произвольные числа в диапазоне от 1 до 10 на основании значения:

```
Math.random() * 10 + 1
```

Если откорректировать метод `Math.random` таким образом, чтобы он всегда выдавал значение 0,5, то "произвольное" число всегда будет равно 6. В результате мы сможем создать модульный тест, как показано в листинге 8.43.

Листинг 8.43. Модульный тест с модифицированным методом `Math.random`

```
it('should tell you that you entered the right answer', async () => {
   // Тест должен сообщить, что введено правильное значение
   // При условии, что приложение было отрисовано
   makeRandomAlways(0.5)
   render(<App />)
   // При вводе правильного ответа
   const input = screen.getByLabelText(/guess:/i)
   const submitButton = screen.getByText('Submit')
   user.type(input, '36')
   user.click(submitButton)
   // Выводится сообщение, что мы выиграли
   await waitFor(() => screen.findByText(/won/i), { timeout: 4000 })
})
```

Этот тест окажется заведомо успешным, поскольку приложение всегда будет задавать вопрос: "Чему равно 6 × 6?"

Настоящая ценность откорректированного метода `Math.random` проявляется при использовании фреймворка для тестирования, который явно предотвращает захват с экрана произвольных значений, как мы видели ранее в случае с фреймворком Cypress.

Но фреймворк Cypress позволяет добавлять пользовательские команды, вставляя их в сценарий cypress/support/commands.js. Отредактируем этот файл, вставив в него код из листинга 8.44.

Листинг 8.44. Код для вставки в файл cypress/support/command.js

```
Cypress.Commands.add('random', (result) => {
   cy.reload().then((win) => {
      if (win.Math.random.restore) {
         win.Math.random.restore()
      }
   sinon.stub(win.Math, 'random').returns(result)
   })
})
```

В результате будет создана новая команда Cypress `cy.random`. Посредством этой команды можно создать тест для выигрышного случая игры, который мы обсуждали во введении этого раздела[17]. Такой тест приведен в листинге 8.45.

Листинг 8.45. Тест для выигрышного случая игры

```
describe('Basic game functions', () => {
   it('should notify the server if I win', () => {
   // Сообщает серверу о выигрыше
      // При условии, что приложение запущено
         cy.intercept('POST', '/api/result', {
         statusCode: 200,
         body: '',
      }).as('postResult')
      cy.visit('http://localhost:3000')
      cy.random(0.5)
      cy.contains('Refresh').click()
      // При вводе правильного ответа
      cy.get('input').type('36')
      cy.contains('Submit').click()
      // Сервер сообщает о выигрыше
      cy.wait('@postResult').then((xhr) => {
         assert.deepEqual(xhr.request.body, {
            guess: 36,
            answer: 36,
            result: 'WIN',
         })
      })
   })
})
```

После вызова команды `cy.random` нужно обновить страницу на случай, если приложение создало произвольные числа до замены метода `Math.random`.

Обсуждение

Полностью удалить из теста все элементы произвольности невозможно. Например, время и частота перерисовки компонентов может в значительной степени зависеть от технических характеристик компьютера. Но устранение как можно большего объема неопределенности в тесте обычно является желаемым результатом. Чем больше внешних зависимостей можно удалить из теста, тем лучше.

Вопрос удаления внешних зависимостей также рассматривается в *разделе 8.9*.

[17] Дополнительная информация об этом тесте приводится в *разделе 8.3*.

Исходный код данного рецепта можно загрузить на веб-сайте GitHub по адресу **https://oreil.ly/P1Tqj**.

8.9. Путешествие во времени

ЗАДАЧА

Время может быть источником огромного количества ошибок. Если бы время было просто предметом научного исследования, тогда работа с ним была бы сравнительно несложной. Но оно таковым не является. Представление времени зависит от государственных границ и местного законодательства. В некоторых странах нет своих собственных часовых поясов, тогда как в других есть несколько таких поясов. Один обнадеживающий фактор в этой мешанине — часовые пояса всех стран смещены от скоординированного всемирного времени UTC на целое число часов. Исключение составляет Индия, часовой пояс которой смещен на +05:30.

По этой причине будет целесообразным использовать в тестах одно постоянное время. Но как это можно сделать?

РЕШЕНИЕ

Рассмотрим, как можно зафиксировать время при тестировании приложения React. При тестировании зависимого от времени кода необходимо учитывать некоторые моменты. Первым делом, скорее всего, не следует изменять время на сервере. В большинстве случаев лучше всего установить на сервере время UTC и больше не менять его.

Это означает, что если в браузере нужно эмулировать дату и время[18], то как только браузер установит контакт с сервером, у нас возникнут проблемы. В результате нам придется или модифицировать API-интерфейс сервера, позволяя ему принимать фактическую дату, или же тестировать зависящий от времени код изолированно от сервера.

Для описанного далее рецепта мы примем второй подход: будем применять систему тестирования Cypress, чтобы эмулировать все подключения к серверу.

Для этого рецепта мы будем пользоваться тем же самым приложением, что и для других примеров в этой главе. Это простая математическая игра, в которой нужно вычислить произведение двух произвольных чисел. Мы будем тестировать возможность игры, которая дает пользователю 30 секунд для ввода ответа. Если в течение 30 секунд пользователь не предоставит никакого ответа, выводится сообщение об истечении времени для ответа (рис. 8.25).

Мы могли бы попробовать создать тест, который каким-либо образом становится на паузу в течение 30 секунд, но с подобным подходом есть две проблемы. Пер-

[18] То есть позволить браузеру сказать серверу: "Давай будем считать, что сегодня вторник, 14 апреля".

вая, — это замедлит тест. Даже несколько 30-секундных пауз сделают процесс тестирования невыносимым. Вторая причина заключается в том, что добавление паузы — это не очень точный способ тестирования возможности. Пауза перед просмотром сообщения с запланированной длительностью 30 секунд может растянуться до 30,5 секунды.

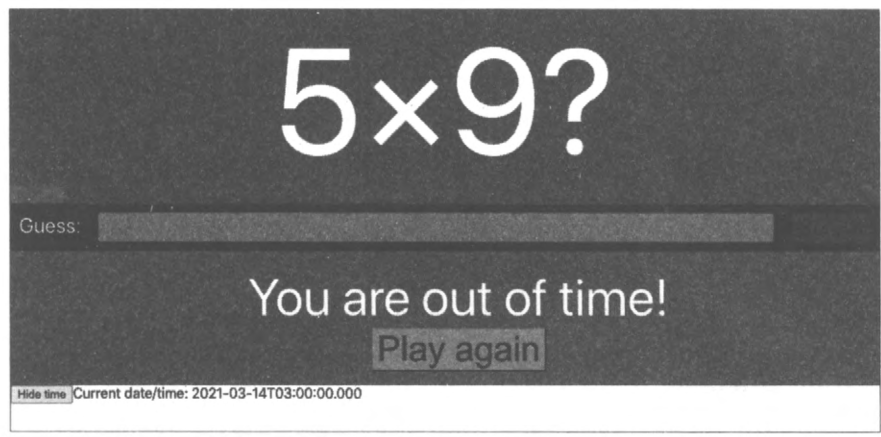


Рис. 8.25. Сообщение об истечении времени для предоставления ответа

Чтобы обеспечить точность, нам нужно управлять временем в браузере. Как мы видели в предыдущем разделе, Cypress может внедрять код в браузер, заменяя критические фрагменты кода методами-заглушками, которыми мы можем управлять. В фреймворке Cypress есть встроенная команда `cy.clock`, которая позволяет задавать текущее время.

Рассмотрим, как задействовать эту команду, создав тест для проверки возможности тайм-аута нашего приложения. Структура теста приведена в листинге 8.46.

Листинг 8.46. Структура теста для проверки возможности тайм-аута

```
describe('Basic game functions', () => {
  it('should say if I timed out', () => {
    // Тест должен сообщить о тайм-ауте
    // При условии, что начата новая игра
    // По истечении 29 секунд
    // Сообщение о тайм-ауте не выводится
    // По истечении еще одной секунды
    // Выводится сообщение о тайм-ауте
    // И игра завершается
  })
})
```

Процесс тестирования можно начать с запуска приложения и нажатия кнопки обновления страницы (листинг 8.47).

Листинг 8.47. Код для запуска процесса тестирования

```
describe('Basic game functions', () => {
   it('should say if I timed out', () => {
      // Тест должен сообщить о тайм-ауте
      // При условии, что начата новая игра
      cy.visit('http://localhost:3000')
      cy.contains('Refresh').click()
      // По истечении 29 секунд
      // Сообщение о тайм-ауте не выводится
      // По истечении еще одной секунды
      // Выводится сообщение о тайм-ауте
      // И игра завершается
   })
})
```

Теперь нам нужно эмулировать отсчет 29 секунд времени. Это можно реализовать при помощи команд `cy.clock` и `cy.tick`. Команда `cy.clock` позволяет задать требуемую новую дату и время; при вызове этой команды без параметров она устанавливает дату 1 января 1970 г. А команда `cy.tick` позволяет добавлять к текущей дате и времени заданное число миллисекунд. Соответствующий код приведен в *листинге 8.48*.

Листинг 8.48. Эмулирование интервала в 29 секунд

```
describe('Basic game functions', () => {
   it('should say if I timed out', () => {
   // Тест должен сообщить о тайм-ауте
      // При условии, что начата новая игра
      cy.clock()
      cy.visit('http://localhost:3000')
      cy.contains('Refresh').click()
      // По истечении 29 секунд
      cy.tick(29000)
      // Сообщение о тайм-ауте не выводится
      // По истечении еще одной секунды
      // Выводится сообщение о тайм-ауте
      // И игра завершается
   })
})
```

Теперь можно создать другие составляющие теста, полный код которого приведен в листинге 8.49. Подробная информация по другим используемым в тесте командам Cypress доступна в документации по этому фреймворку по адресу **https://oreil.ly/vahMA**.

Листинг 8.49. Полный тест проверки тайм-аута

```
describe('Basic game functions', () => {
   it('should say if I timed out', () => {
      // Тест должен сообщить о тайм-ауте
      // При условии, что начата новая игра
      cy.clock()
      cy.visit('http://localhost:3000')
      cy.contains('Refresh').click()
      // По истечении 29 секунд
      cy.tick(29000)
      // Сообщение о тайм-ауте не выводится
      cy.contains(/out of time/i).should('not.exist')
      // По истечении еще одной секунды
      cy.tick(1000)
      // Выводится сообщение о тайм-ауте
      cy.contains(/out of time/i).should('be.visible')
      // И игра завершается
      cy.get('input').should('be.disabled')
      cy.contains('Submit').should('be.disabled')
   })
})
```

При исполнении этого теста в Cypress он успешно выполняется, как можно видеть на рис. 8.26.

Рис. 8.26. Управляя временем, можно принудительно создать тайм-аут в тесте

Это был сравнительно простой тест с управлением временем. А как быть, если мы хотим протестировать что-либо намного более сложное, например летнее время?

Ошибки, связанные с этим временем, являются бичом большинства разработчиков. Они "молча" сидят в коде в течение долгих месяцев, а потом внезапно врываются в жизнь ранним весенним или осенним утром.

Возникновение ошибок, связанных с летним временем, зависит от конкретного часового пояса. С ними особенно трудно бороться в клиентском коде, поскольку даты в JavaScript не поддерживают часовых поясов. Они, конечно же, могут справиться со смещениями. Например, в браузере Chrome можно создать объект `Date` со значением на пять часов раньше времени по Гринвичу[19]:

```
new Date('2021-03-14 01:59:30 GMT-0500')
```

Но даты JavaScript неявно находятся в часовом поясе используемого браузера. При создании даты, содержащей название часового пояса, движок JavaScript просто сместит ее значение в часовой пояс браузера.

А часовой пояс браузера фиксируется при запуске браузера, и не существует никакого способа сказать: "Давай будем считать, что с этого момента мы будем в Нью-Йорке".

Может случиться так, что тесты для летнего времени будут работать только в часом поясе разработчика, а при исполнении на сервере интеграции, для которого установлено скоординированное всемирное время, будут завершаться сбоем.

Но существует способ обойти эту проблему. На компьютерах Linux и Mac (но не Windows) при запуске браузера можно указать требуемый часовой пояс, присвоив соответствующее значение переменной среды `TZ`. Если запустить фреймворк Cypress с установленным значением этой переменной, то любой запускаемый в Cypress браузер унаследует это значение. В итоге, несмотря на то, что мы не можем задать часовой пояс для отдельного теста, можно это сделать для всего тестового прогона.

Запустим Cypress с часовым поясом, заданным для Нью-Йорка:

```
$ TZ='America/New_York' npx cypress open
```

В окне нашего приложения есть кнопка, которая позволят отображать или скрывать текущее время (рис. 8.27).

Давайте создадим тест, проверяющий правильность перехода времени на странице на летнее время. Код теста приведен в листинге 8.50.

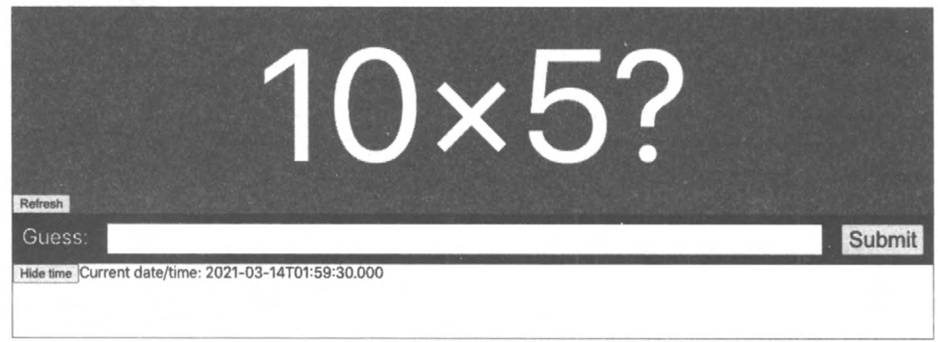


Рис. 8.27. Отображение текущего времени в окне приложения

[19] Браузер Firefox обычно не принимает этот формат.

Листинг 8.50. Код теста для проверки перехода на летнее время

```
describe('Timing', () => {
  it('should tell us the current time', () => {
  // Тест должен сообщить текущее время
    cy.clock(new Date('2021-03-14 01:59:30').getTime())
    cy.visit('http://localhost:3000')
    cy.contains('Show time').click()
    cy.contains('2021-03-14T01:59:30.000').should('be.visible')
    cy.tick(30000)
    cy.contains('2021-03-14T03:00:00.000').should('be.visible')
  })
})
```

В тесте мы передаем команде `cy.clock` явную дату. Значение этой даты нужно преобразовать в миллисекунды, вызывая метод `get.Time`, поскольку `cy.clock` может принимать только числовые значения. Затем мы проверяем начальное время, а по истечении 30 секунд проверяем переход времени на 03:00 вместо 02:00, как показано на рис. 8.28.

Рис. 8.28. По истечении 30 секунд время правильно меняется с 01 59 на 03·00

Обсуждение

Тесты, зависящие от текущего часового пояса, целесообразно поместить в подпапку, чтобы их можно было исполнять отдельно. Для форматирования дат в разные часовые пояса можно использовать метод `toLocaleString`:

```
new Date().toLocaleString('en-US', { timeZone: 'Asia/Tokyo' })
```

Исходный код данного рецепта можно загрузить на веб-сайте GitHub по адресу **https://oreil.ly/P1Tqj**.

ГЛАВА 9

Доступность специальных возможностей

Написание этой главы было сопряжено с определенными трудностями, поскольку кроме очков или контактных линз никто из нас для работы с компьютером не нуждается в каком-либо специальном оборудовании или программном обеспечении. В этой главе мы попытались собрать коллекцию инструментов и методов, которые в идеале помогут вам найти некоторые из наиболее очевидных проблем с реализацией специальных возможностей в вашем коде.

Мы рассмотрим, как можно использовать ориентиры и роли ARIA, добавляющие значение и структуру в страницы приложения, которые в противном случае имели бы лишь визуальную группировку. Затем в нескольких рецептах мы продемонстрируем, как исполнять ручные и автоматизированные аудиты приложения, искать глюки в коде посредством статического анализа, а также находить ошибки времени исполнения, автоматизируя браузеры.

Далее мы рассмотрим некоторые технические моменты, связанные с созданием пользовательских диалоговых окон (подсказка: старайтесь применять готовые диалоговые окна, предоставляемые библиотеками) и, наконец, создадим простую программу для чтения текста с экрана.

Для более подробного рассмотрения темы доступности специальных возможностей обязательно посетите веб-сайт WCAG (Web Content Accessibility Guidelines — нормы доступности к веб-содержимому, **https://oreil.ly/ie0aT**), на котором определяются три уровня соответствия этим нормам: A, AA и AAA. Самый высший уровень соответствия — AAA.

Для разработчиков профессионального программного обеспечения эти рецепты будут, скорее всего, полезными. Но ничто не сможет заменить опыт тех, кому каждый день приходится преодолевать препятствия, создаваемые недоступным программным обеспечением. Доступное программное обеспечение — это просто хорошее программное обеспечение. Оно максимизирует вашу рыночную долю и заставляет вас уделять больше внимания дизайну. Мы бы порекомендовали как минимум выполнять аудит доступности своего кода. Необходимые для этого инструменты можно получить, связавшись с организацией наподобие AbilityNet в Великобритании (**https://oreil.ly/N7XkH**) или просто поискав программное обеспечение для тестирования доступности по своему местонахождению. Вы увидите, что это наиболее эффективный способ для обнаружения проблем доступности в разрабатываемом коде.

9.1. Использование ориентиров

ЗАДАЧА

Рассмотрим простое приложение для создания и управления задачами, показанное на рис. 9.1.

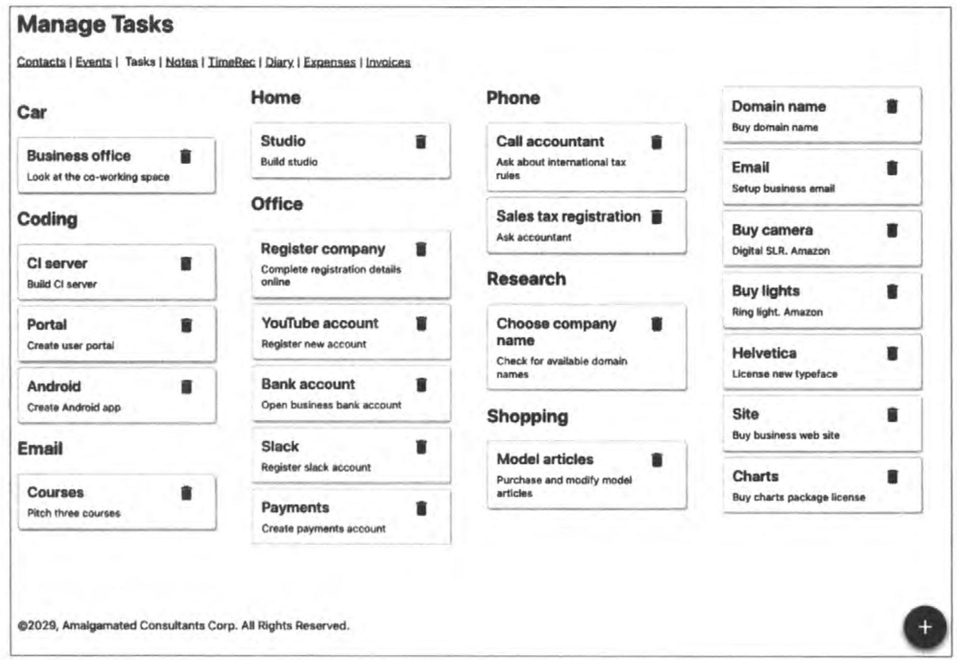


Рис. 9.1. Окно примера приложения для создания и управления задачами

Пользователь, который увидит окно этого приложения, с легкостью различит основное содержимое (задачи) и все прочие элементы по краям: ссылки на другие страницы, заголовки, заявление об авторском праве и т. п.

Давайте рассмотрим код основного компонента App этого приложения (листинг 9.1).

Листинг 9.1. Код компонента App примера приложения

```
const App = () => {
  ...
  return (
    <>
    <h1>Manage Tasks</h1>
    <a href='/contacts'>Contacts</a> | 
    <a href='/events'>Events</a> | 
    Tasks | 
    <a href='/notes'>Notes</a> | 
    <a href='/time'>TimeRec</a> | 
    <a href='/diary'>Diary</a> | 
```

```
        <a href='/expenses'>Expenses</a> | 
        <a href='/invoices'>Invoices</a>
        <button className='addButton'
           onClick={() => setFormOpen(true)}>+</button>
        <TaskContexts .../>
        &#169;2029, Amalgamated Consultants Corp. All Rights Reserved.
        <TaskForm .../>
        <ModalQuestion ...>
           Are you sure you want to delete this task?
        </ModalQuestion>
     </>
   )
}
```

Проблема с этим кодом состоит в том, что при чтении страницы посредством какого-либо устройства определить ее структуру трудно. Какие из ее составляющих являются навигационными ссылками? Где основное содержимое страницы? Такой парсинг страницы, который с легкостью осуществляет человеческий глаз (рис. 9.2), невозможно осуществить без доступа к пространственному группированию интерфейса.

Так как же решить эту проблему? Что можно использовать вместо визуального группирования, чтобы сделать структуру страницы более понятной?

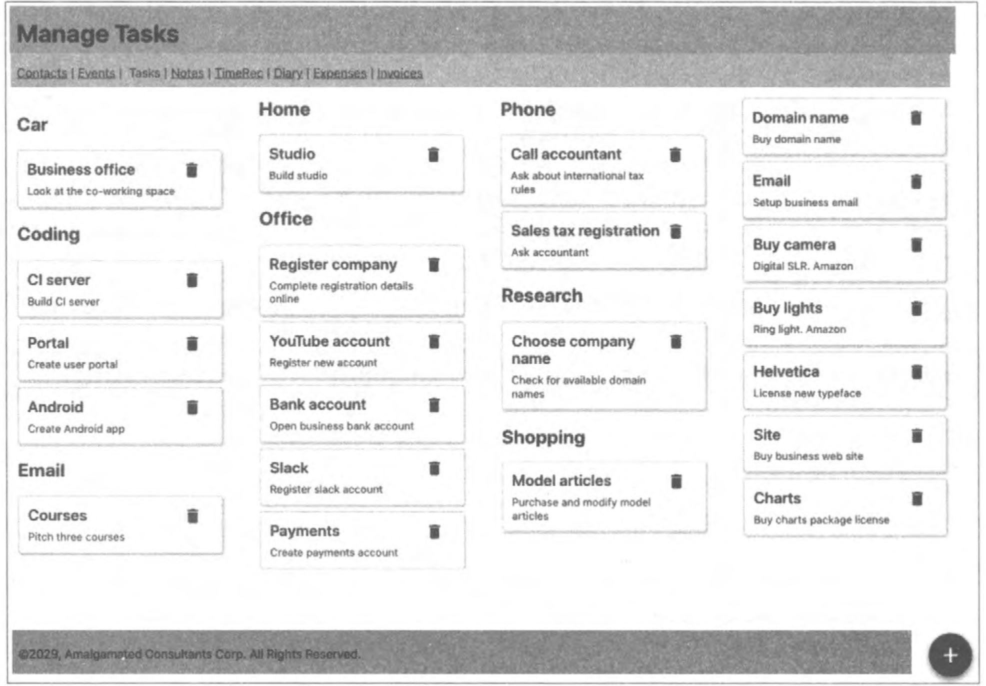


Рис. 9.2. Обычные пользователи могут с легкостью определить пространственное местонахождение разделов страницы

РЕШЕНИЕ

Для решения этой задачи мы будем использовать ориентиры в нашем коде. Ориентиры — это элементы HTML, посредством которых можно структурно сгруппировать части интерфейса, чтобы отобразить их визуальную группировку. Ориентиры также полезны при разработке страницы, поскольку они заставляют разработчика думать о назначении разных типов содержимого.

Начнем с выделения *заголовка.* Эта часть страницы определяет ее содержимое. Обычно используется заголовок h1, но можно также включить широко применяемые инструменты или, возможно, логотип. Идентифицировать заголовок можно с помощью тега <header>:

```
<header>
  <h1>Manage Tasks</h1>
</header>
```

Страница всегда должна содержать заголовок h1, а заголовки более низких уровней должны обеспечивать структурирование содержимого остальной части страницы, не пропуская при этом никаких уровней. Например, никогда нельзя указывать заголовки h1 и h3 без заголовка h2 где-либо между этими двумя заголовками. Ориентиры являются удобным навигационным методом для людей, пользующихся средствами чтения с экрана, включая функции, которые позволяют пользователям переходить вперед и назад между заголовками.

Далее нам нужно подумать о средствах навигации по странице. Они могут быть разной формы, например список ссылок (как в данном случае), или последовательность меню или боковая колонка. Средства навигации — это блок компонентов, которые позволяют посещать основные части веб-сайта. Почти наверняка страница будет содержать другие ссылки, не относящиеся к средствам навигации.

Для идентификации средств навигации на нашей странице подойдет ориентир в виде тега <nav>, как показано в листинге 9.2.

Листинг 9.2. Выделение средств навигации при помощи тега <nav>

```
<nav>
  <a href='/contacts'>Contacts</a> | 
  <a href='/events'>Events</a> | 
  Tasks | 
  <a href='/notes'>Notes</a> | 
  <a href='/time'>TimeRec</a> | 
  <a href='/diary'>Diary</a> | 
  <a href='/expenses'>Expenses</a> | 
  <a href='/invoices'>Invoices</a>
</nav>
```

Содержимое страницы является ее основной частью. В нашем приложении по созданию и управлению задачами содержимое состоит из коллекции задач. Основным

содержимым будет то, что пользователь хочет читать и с чем он хочет взаимодействовать на странице в первую очередь. Иногда основное содержимое может также содержать инструменты, например плавающую кнопку для добавления задач. Но они не обязательно должны находиться в основном содержимом, и их можно переместить куда-либо в заголовок.

Основное содержимое страницы можно сгруппировать при помощи тега `<main>`, как показано в листинге 9.3.

Листинг 9.3. Группирование содержимого посредством тега `<main>`

```
<main>
   <button className='addButton'
      onClick={() => setFormOpen(true)}>+</button>
   <TaskContexts contexts={contexts}
      tasks={tasks}
      onDelete={setTaskToRemove}
      onEdit={task => {
         setEditTask(task)
         setFormOpen(true)
      }}
   />
</main>
```

Наконец, есть метаданные веб-страницы: данные о данных. Примером метаданных является сообщение о защите авторских прав в нашем приложении. Метаданные часто помещаются внизу страницы, сгруппированные посредством тега `<footer>`:

```
<footer>
   &#169;2029, Amalgamated Consultants Corp. All Rights Reserved.
</footer>
```

Осталось рассмотреть еще два раздела исходного компонента `App`: `TaskForm` и `ModalQuestion`. В листинге 9.4 приведен код этих разделов.

Листинг 9.4. Код разделов `TaskForm` и `ModalQuestion`

```
<TaskForm .../>
<ModalQuestion ...>
   Are you sure you want to delete this task?
</ModalQuestion>
```

Раздел `TaskForm` — это модальное окно, которое открывается, когда пользователь хочет создать или редактировать задачу (рис. 9.3).

А раздел `ModalQuestion` представляет собой модальное окно подтверждающего сообщения, отображающегося, когда пользователь пытается удалить задачу (рис. 9.4).

Рис. 9.3. Модальное окно **TaskForm** отображается поверх всего другого содержимого окна приложения

Рис. 9.4. Модальное окно **ModalQuestion** открывается, чтобы подтвердить удаление задачи пользователем

Эти два компонента отображаются только тогда, когда они требуются. При нормальном состоянии страницы эти модальные разделы не входят в ее структуру, поэтому ориентиры для них не требуются. Далее в этой главе мы рассмотрим другие подходы к работе с динамическим содержимым, таким как модальные окна, которые делают их более доступными для пользователей приложения.

Конечная модифицированная версия компонента `App` приведена в листинге 9.5.

Листинг 9.5. Конечная модифицированная версия компонента App

```
const App = () => {
  ....
  return (
    <>
      <header>
        <h1>Manage Tasks</h1>
      </header>
      <nav>
        <a href='/contacts'>Contacts</a> | 
        <a href='/events'>Events</a> | 
        Tasks | 
        <a href='/notes'>Notes</a> | 
        <a href='/time'>TimeRec</a> | 
        <a href='/diary'>Diary</a> | 
```

```
<a href='/expenses'>Expenses</a> | 
          <a href='/invoices'>Invoices</a>
        </nav>
        <main>
        <button className='addButton'
          onClick={() => setFormOpen(true)}>+</button>
          <TaskContexts .../>
        </main>
        <footer>
          &#169;2029, Amalgamated Consultants Corp. All Rights Reserved.
        </footer>
        <TaskForm .../>
        <ModalQuestion ...>
          Are you sure you want to delete this task?
        </ModalQuestion>
      </>
  )
}
```

Обсуждение

Ориентиры являются частью HTML5, поэтому их поддержка встроена в браузеры. Это означает, что приступать к их использованию можно без установки каких бы то ни было специальных инструментов или библиотек поддержки.

Некоторым автоматизированным средствам доступности могут "не нравиться" ориентиры, отрисованные приложениями React. Согласно стандартным нормам все содержимое тела веб-страницы должно быть заключено в тег ориентира. Но большинство приложений React отрисовывают свое содержимое (включая ориентиры) внутри одного тега `<div>`, что сразу же нарушает все правила.

Скорее всего, эту проблему можно игнорировать, не опасаясь каких-либо отрицательных последствий. Если все имеющиеся ориентиры одного уровня, то факт их заключения в дополнительный тег `<div>` не должен играть никакой роли.

Исходный код данного рецепта можно загрузить на веб-сайте GitHub по адресу **https://oreil.ly/0GfgA**.

9.2. Применение ролей

ЗАДАЧА

В приложениях часто встречаются компоненты, которые ведут себя как кнопки, хотя и не являются кнопками. Подобным образом в приложениях могут использоваться компоненты, которые выглядят как всплывающие диалоговые окна, не являясь в действительности таковыми. Также приложения могут содержать коллекции данных, структура которых похожа на списки, но без наличия тегов `<ol>` или `<ul>`.

Создание компонентов, которые ведут себя подобно стандартным элементам пользовательского интерфейса, не представляет никаких проблем, если пользователи могут видеть визуальное оформление компонентов. Если какой-либо элемент выглядит похожим на кнопку, пользователь будет обращаться с ним, как с кнопкой, независимо от способа его реализации.

Но с этим возникает проблема, если пользователи не могут увидеть визуального оформления компонента. Для этих пользователей нужно описывать назначение таких компонентов.

РЕШЕНИЕ

Данная задача решается посредством использования в приложении *ролей. Роль* описывает значение компонента, сообщая пользователю о его назначении. Роли входят в состав семантики веб-страницы и поэтому похожи на семантические ориентиры, рассматриваемые в *разделе 9.1*.

В *табл. 9.1* приведен список типичных ролей, которые можно применять к отрисованным элементам HTML.

Таблица 9.1. *Список типичных ролей и их краткое описание*

Название роли	**Назначение**
`alert`	Сообщает пользователю, что что-то произошло
`article`	Большой блок текстового содержимого, наподобие новостной заметки
`button`	Объект, по которому можно щелкнуть, чтобы выполнить какое-либо действие
`checkbox`	Переключатель со значением истина/ложь, устанавливаемым пользователем
`comment`	Наподобие отправляемого пользователем комментария или отзыва
`complementary`	Дополнительная информация, возможно, в боковой панели
`contentinfo`	Сообщения о защите авторских прав, имена авторов, даты издания
`dialog`	Какой-либо объект, всплывающий поверх другого содержимого. Часто модальный
`feed`	Список статей. Широко используется в блогах
`figure`	Иллюстрация
`list`	Список объектов
`listitem`	Элемент списка
`search`	Поле поиска
`menu`	Набор опций меню, обычно используемых для навигации
`menuitem`	Элемент меню

Роли применяются к элементам при помощи атрибута `role`. Рассмотрим компонент `Task` из примера приложения для этой главы. Данный компонент отрисовывает каж-

дую задачу в виде небольшой панельки, содержащей кнопку удаления в виде корзины. В листинге 9.6 приведен код этого компонента.

Листинг 9.6. Код компонента Task

```
import DeleteIcon from './delete-24px.svg'
import './Task.css'
const Task = ({ task, onDelete, onEdit }) => {
   return (
      <div className="Task">
         <div className="Task-contents"
            ...
         >
         <div className="Task-details">
            <div className="Task-title">{task.title}</div>
            <div className="Task-description">{task.description}</div>
         </div>
         <div className="Task-controls">
            <img
               src={DeleteIcon}
               width={24}
               height={24}
               title="Delete"
               onClick={(evt) => {
                  evt.stopPropagation()
                  onDelete()
               }}
               alt="Delete icon"
               />
            </div>
         </div>
      </div>
   )
}
```

Задачи группируются на странице под заголовками, описывающими контекст, в котором они выполняются. Например, несколько задач могут быть сгруппированы под заголовком Phone (рис. 9.5).

Таким образом, похоже, что задачам соответствует роль элемента списка `listitem`. Это объекты, которые находятся внутри упорядоченной коллекции. Поэтому первому разделу `<div>` можно присвоить эту роль:

```
return <div role='listitem' className='Task'>
   <div className='Task-details'>
      ....
```

Но если остановиться на этом, то у нас возникнет проблема. Роли должны подчиняться определенным правилам. В частности, роль `listitem` можно присвоить толь-

ко компонентам, которые находятся внутри объекта с ролью `list`. Поэтому, чтобы компонентам `Task` можно было присвоить роль `listitem`, их родительскому компоненту `TaskList` необходимо присвоить роль `list`. Соответствующий код приведен в листинге 9.7.

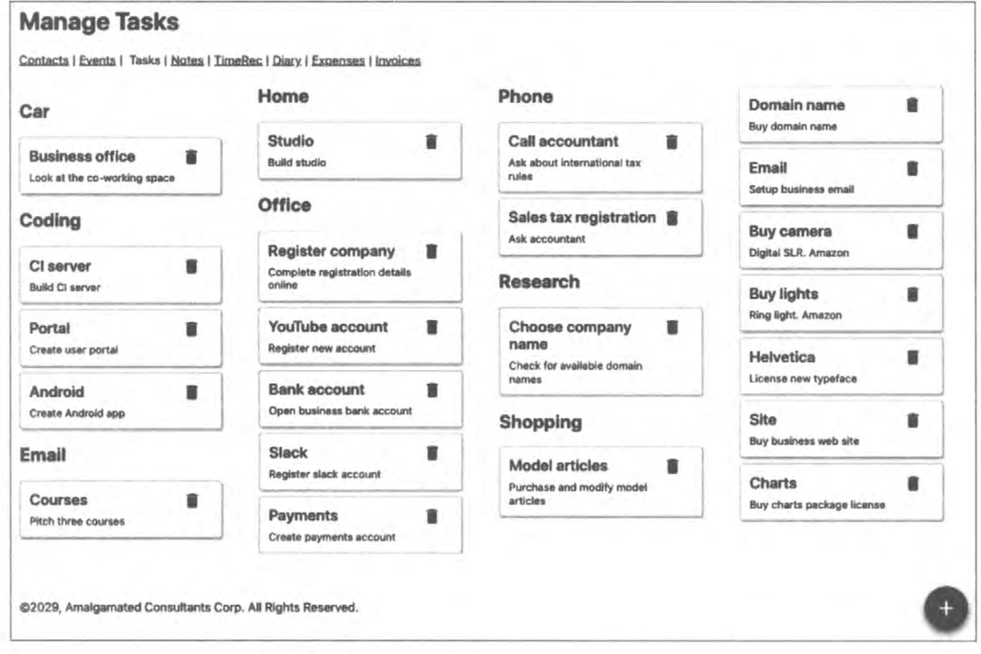


Рис. 9.5. Каждая группа содержит список задач

Листинг 9.7. Присвоение роли `list` компоненту `TaskList`

```
import Task from '../Task'
import './TaskList.css'

function TaskList({ tasks, onDelete, onEdit }) {
   return (
      <div role="list" className="TaskList">
         {tasks.map((t) => (
            <Task
               key={t.id}
               task={t}
               onDelete={() => onDelete(t)}
               onEdit={() => onEdit(t)}
            />
         ))}
      </div>
   )
}
export default TaskList
```

Применение ролей list и listitem полностью допустимо. Но, наверное, если наш HTML ведет себя подобно списку, на практике будет намного лучше изменить разметку и использовать настоящие теги <ul> и <li>. С точки зрения доступности это, скорее всего, не будет иметь никакого значения. Но всегда лучше избегать заполнения кода HTML бесконечными тегами <div>. В общем, если роль можно заменить обычным тегом HTML, то, видимо, лучше всего так и сделать.

Поэтому давайте удалим из компонента TaskList роль list и добавим настоящий тег <ul>, как показано в листинге 9.8.

**Листинг 9.8. Замена роли list тегом **

```
import Task from '../Task'
import './TaskList.css'
function TaskList({ tasks, onDelete, onEdit }) {
   return (
      <ul className="TaskList">
         {tasks.map((t) => (
            <Task
               key={t.id}
               task={t}
               onDelete={() => onDelete(t)}
               onEdit={() => onEdit(t)}
            />
         ))}
      </ul>
   )
}
export default TaskList
```

А затем в компоненте Task заменим роль listitem тегом <li>, как показано в листинге 9.9.

**Листинг 9.9. Замена роли listitem тегом **

```
import './Task.css'
const Task = ({ task, onDelete, onEdit }) => {
   return (
      <li className="Task">
         <div
            className="Task-contents"
            ...
         >
            <div className="Task-details">...</div>
            <div className="Task-controls">...</div>
         </div>
```

```
        </li>
    )
}
export default Task
```

Использование тегов <li> означает, что нам нужно изменить несколько стилей CSS, чтобы удалить маркеры элементов списка, но такой код будет легче читать любому разработчику (которым можете оказаться вы), которому придется работать с ним в будущем.

Далее рассмотрим раздел навигации нашего примера приложения. Он содержит ряд ссылок, которые можно рассматривать, как меню опций (листинг 9.10).

Листинг 9.10. Раздел навигации примера приложения

```
<nav>
   <a href='/contacts'>Contacts</a> | 
   <a href='/events'>Events</a> | 
   Tasks | 
   <a href='/notes'>Notes</a> | 
   <a href='/time'>TimeRec</a> | 
   <a href='/diary'>Diary</a> | 
   <a href='/expenses'>Expenses</a> | 
   <a href='/invoices'>Invoices</a>
</nav>
```

Следует ли использовать здесь роли menu и menuitem? Ответ на этот вопрос — почти однозначно нет.

Меню и элементы меню подразумевают ожидаемое поведение. Пользователь, желающий вызвать меню, будет, наверное, ожидать, что оно раскроется при выборе. А в раскрывшемся меню он, наверное, будет перемещаться по опциям посредством клавиш стрелок, а не клавиши <Tab>[1].

Теперь рассмотрим кнопку + в правом нижнем углу приложения, которая позволяет пользователю создавать новое задание, открывая соответствующую всплывающую форму (рис. 9.6).

Код для этой кнопки следующий:

```
<button className='addButton'
   onClick={() => setFormOpen(true)}>+</button>
```

Нужно ли присвоить данной кнопке какую-либо роль? Нет, не нужно, поскольку этот элемент уже и так является кнопкой. Но мы в состоянии предоставить некоторую дополнительную информацию о том, что может ожидать пользователь, когда он нажмет эту кнопку. Как уже упоминалось чуть ранее, нажатие этой кнопки

[1] В интересной статье Адриана Роселли (Adrian Roselli, **https://oreil.ly/i8AMI**) рассматриваются вопросы, касающиеся меню и элементов меню.

вызывает открытие всплывающего окна. Такое действие можно указать явно в коде HTML посредством атрибута `aria-haspopup`:

```
<button aria-haspopup='dialog' className='addButton'
   onClick={() => setFormOpen(true)}>+</button>
```

Рис. 9.6. Нажатие кнопки + открывает всплывающую форму для создания новой задачи

Значение атрибута `aria-haspopup` должно совпадать с ролью компонента, который отобразится в результате нажатия кнопки. В данном случае мы будем отображать диалоговое окно. Атрибуту `aria-haspopup` также можно присвоить значение `true`. Тем не менее средство чтения с экрана будет интерпретировать этот компонент как `menu`, поскольку компоненты со связанными с ними всплывающими окнами обычно служат для открытия меню.

Поскольку мы присвоили атрибуту `aria-haspopup` значение `dialog`, нам нужно обеспечить, чтобы роль отображаемого компонента `TaskForm` также была `dialog`. В листинге 9.11 приведен текущий код компонента `TaskForm`.

Листинг 9.11. Текущий код компонента `TaskForm`

```
const TaskForm = ({ task, contexts, onCreate, onClose, open }) => {
   ...
   return <Modal open={open} onCancel={close}>
      <form>
         ....
      </form>
```

```
      <ModalFooter>
         <button onClick={...}>Cancel</button>
         <button onClick={...}>Save</button>
      </ModalFooter>
   </Modal>
}
```

Мы заключим компонент `TaskForm` в компонент `Modal`, как показано в листинге 9.12.

Листинг 9.12. Заключение компонента `TaskForm` в компонент `Modal`

```
import './Modal.css'
function Modal({ open, onCancel, children }) {
   if (!open) {
      return null
   }

   return <div className='Modal'
      ...
   >
      <div className='Modal-dialog'
         ...
      >
         {children}
      </div>
   </div>
}
export default Modal
```

Данный компонент `Modal` состоит из двух частей:

- внешняя оболочка `Modal`, которая служит для затенения другого содержимого страницы и является полупрозрачным слоем;
- внутренний раздел `Modal-dialog`, который отображает содержимое в контейнере, имеющем вид окна.

Класс `Modal` допускает многократное использование и пригоден для других объектов, например для окон предупреждения. Поэтому мы присвоим ему дополнительное свойство `title`, которое будет применяться к компоненту `Modal-dialog`. Благодаря свойству `title` назначение данного диалогового окна будет понятным для любого пользователя.

Модифицированный таким образом компонент `Modal` приведен в листинге 9.13.

Листинг 9.13. Код модифицированного компонента `Modal`

```
import './Modal.css'
function Modal({ open, onCancel, children, role, title }) {
```

```
   if (!open) {
      return null
}
   return <div role='presentation' className='Modal'
         ...
   >
      <div className='Modal-dialog'
         role={role} title={title}
         ...
      >
         {children}
      </div>
   </div>
}
export default Modal
```

А в листинге 9.14 приведен код модифицированного компонента TaskForm.

Листинг 9.14. Код модифицированного компонента TaskForm

```
const TaskForm = ({ task, contexts, onCreate, onClose, open }) => {
   ...
   return <Modal title='Create or edit a task'
         role='dialog'
         open={open} onCancel={close}>
      <form>
         ....
      </form>
      <ModalFooter>
         <button onClick={...}>Cancel</button>
         <button onClick={...}>Save</button>
      </ModalFooter>
   </Modal>
}
```

Наконец, рассмотрим кнопку удаления задачи в виде значка мусорной корзины, расположенного рядом с каждой задачей (листинг 9.15).

Листинг 9.15. Код кнопки удаления задачи

```
<img src={DeleteIcon}
   width={24}
   height={24}
   alt='Delete icon'
   aria-haspopup='dialog'
   role='button'
   title='Delete'
```

```
   onClick={evt => {
      evt.stopPropagation()
      evt.preventDefault()
      onDelete()
   }}
/>
```

Значок мусорной корзины функционирует как кнопка, поэтому мы и присвоили ему эту роль. Он уже имеет атрибут `aria-haspopup`, поскольку в диалоговом окне пользователю даются указания подтвердить удаление.

Но точно так же, как и в случае со списками и элементами списка, лучше реализовать компоненты, функционирующие как кнопки, в виде настоящих кнопок HTML. Поэтому мы превратим этот компонент в кнопку, облекающую изображение, как показано в листинге 9.16.

Листинг 9.16. Модифицированный код кнопки удаления задачи

```
<button
   onClick={evt => {
      evt.stopPropagation()
      evt.preventDefault()
      onDelete()
   }}
   title='Delete'
   aria-haspopup='dialog'
>
   <img src={DeleteIcon}
      width={24}
      height={24}
      alt='Delete icon'
   />
</button>
```

Такой код будет не только более понятным для разработчиков, но также позволит перемещаться к этой кнопке и устанавливать на ней фокус посредством клавиши <Tab>.

Обсуждение

В некоторых аспектах роли совмещаются с ориентирами. Например, существуют роли для ориентиров, такие как `main` и `header`. Но предназначение ролей и ориентиров разное. Ориентиры, как можно предполагать по самому названию, являются способом для выделения основных частей веб-страницы. Роли же, наоборот, описывают предполагаемое поведение какой-либо части интерфейса. В обоих случаях как ориентиры, так и роли служат для придания дополнительной информативности веб-странице.

Если интерфейс содержит компоненты, функционирующие как стандартные элементы HTML (например, списки), часто будет лучше оформить соответствующий стиль обычной HTML-разметкой, чем создавать элементы, кодируя их самостоятельно.

Исходный код данного рецепта можно загрузить на веб-сайте GitHub по адресу **https://oreil.ly/0GfgA**.

9.3. Проверка доступности посредством ESlint

ЗАДАЧА

Когда нет необходимости в оборудовании для обеспечения доступности, идентификация проблем с доступностью может представлять трудности[2]. В процессе разработки также могут возникнуть проблемы регрессии, нарушающие доступность ранее протестированного кода.

Что нам нужно, так это способ, позволяющий быстро и легко обнаружить проблемы с доступностью по мере их возникновения. Нам требуется какой-либо процесс, который непрерывно наблюдает за создаваемым кодом и немедленно фиксирует возникающие проблемы, пока мы еще помним, что этот код делает.

РЕШЕНИЕ

Более очевидные проблемы с доступностью в коде можно обнаружить при помощи инструмента `eslint`, сконфигурированного должным образом.

Инструмент `eslint` выполняет статический анализ кода. Он может найти неиспользуемые переменные, пропущенные зависимости в вызовах хука `useEffect` и т. п. Если вы создали свое приложение при помощи средства `create-reat-app`, то, скорее всего, средство `eslint` уже постоянно исполняется для вашего приложения. Сервер для разработки запускает `eslint` на исполнение при каждом компилировании кода, и любые ошибки `eslint` отображаются в окне сервера.

Если средство `eslint` еще не установлено на вашем компьютере, его можно установить, выполнив следующую команду `npm`:

```
$ npm install --save-dev eslint
```

Точно так же `eslint` можно установить и при помощи `yarn`. Возможности `eslint` можно расширить с помощью подключаемых модулей. Подключаемый модуль для `eslint` содержит набор правил, соблюдение которых статическим кодом `eslint` проверяет при сохранении кода. Один из подключаемых модулей специально предназначен для тестирования на наличие проблем с доступностью. Этот модуль называется `jsx-a11y` и устанавливается следующим образом:

```
$ npm install --save-dev eslint-plugin-jsx-a11y
```

---

[2] Мы обнаружили это сами в процессе работы над данной главой. В результате мы, несомненно, пропустили много, много проблем с доступностью в примере приложения.

Средство `eslint` можно исполнять вручную, добавив в файл package.json приложения код, приведенный в листинге 9.17[3].

Листинг 9.17. Код для ручного исполнения `eslint`

```
"scripts": {
  ....
  "lint": "eslint src"
},
```

Но прежде чем использовать модуль `jsx-a11y`, его нужно сконфигурировать. Это делается путем обновления раздела `eslintConfig` файла package.json, как показано в листинге 9.18.

Листинг 9.18. Обновление раздела `eslintConfig`

```
"eslintConfig": {
  "extends": [
    "react-app"
    "react-app/jest",
    "plugin:jsx-a11y/recommended"
  ],
  "plugins": [
    "jsx-a11y"
  ],
  "rules": {}
}
```

Эта конфигурация дает указание `eslint` применить новый модуль, а также подключает набор рекомендуемых правил по доступности.

При желании можно также настроить работу каждого из правил, добавив дополнительные конфигурационные данные в раздел `rules`. В листинге 9.19 приведен пример такой настройки, с отключением одного из правил.

Листинг 9.19. Отключение правила `eslint`

```
"eslintConfig": {
  "extends": [
    "react-app"
    "react-app/jest",
    "plugin:jsx-a11y/recommended"
  ],
```

[3] Это особенно полезно, если нужно проверить код в хуках `pre-commit` Git или на сервере интеграции.

```
  "plugins": [
  "jsx-a11y"
  ],
  "rules": {
    "jsx-a11y/no-onchange": "off"
  }
}
```

Отключение всех правил нецелесообразно, но для отключения конкретного правила no-onchange может быть веская причина.

Разработчики модуля jsx-a11y создали это правило из-за проблемы со старыми браузерами, каждый из которых реализовывал правило onchange по-своему. Одни браузеры генерировали событие onChange всякий раз, когда пользователь вводил символ в поле ввода, а другие создавали это событие только после того, как пользователь покинул это поле. Такое разное поведение создавало множество проблем для людей, работающих со средствами обеспечения доступности.

Возможное решение — заменить все обработчики события onChange обработчиками события onBlur, тогда все браузеры создавали бы событие изменения поля единообразно: по выходу пользователя из поля.

Но в настоящее время данное правило полностью устарело, и поэтому его не рекомендуется использовать в модуле jsx-a11y. Если в коде React заменить все обработчики события onChange обработчиками события onBlur, то приложение будет работать совсем по-другому. Кроме того, в результате изменится стандартный способ отслеживания состояния полей форм в React: через событие onChange.

Поэтому в данном случае лучше отключить это правило.

Теперь, разрешив правила доступности, можно запустить eslint на исполнение:

```
$ npm run lint
```

В одной из ранних версий приложения eslint обнаружил несколько ошибок (листинг 9.20).

Листинг 9.20. Ошибки, обнаруженные eslint в ранней версии приложения

```
$ npm run lint
> app@0.1.0 lint app
> eslint src
app/src/Task/Task.js
   6:9 error Visible, non-interactive elements with click handlers
             must have at least one keyboard listener
             jsx-a11y/click-events-have-key-events
          (Видимые, неинтерактивные элементы с обработчиками щелчков
           должны иметь, по крайней мере, один слушатель клавиатуры)
  6:9 error Static HTML elements with event handlers require a role
 (Статические HTML-элементы с обработчиками событий должны иметь роль)
            jsx-a11y/no-static-element-interactions
✖ 2 problems (2 errors, 0 warnings)
```

Чтобы разобраться, в чем причина этих ошибок, взглянем на исходный код в файле Task.js (листинг 9.21).

Листинг 9.21. Исходный код компонента `Task`

```
<li className="Task">
  <div className="Task-contents" onClick={onEdit}>
    ....
  </div>
</li>
```

Компонент `Task` отображает подробности задачи внутри небольшой панели, называющейся карточкой (рис. 9.7).

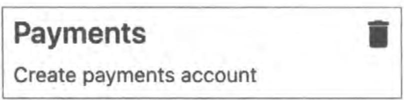


Рис. 9.7. Приложение отображает задачи в отдельных панелях, каждая из которых содержит кнопку удаления задачи

Щелчок мышью по панели задачи открывает форму для редактирования подробностей задачи. Это осуществляется посредством обработчика события `onClick` для раздела `<div>` `Task-contents`.

Чтобы понять, что здесь "не нравится" `eslint`, сначала посмотрим на следующее сообщение об ошибке:

```
6:9 error Static HTML elements with event handlers require a role
(Статические HTML-элементы с обработчиками событий должны иметь роль)
        jsx-a11y/no-static-element-interactions
```

Такие элементы, как `<div>`, являются статическими и не имеют встроенного интерактивного поведения. По умолчанию, это просто объекты, которые упорядочивают другие объекты. Недовольство `eslint` вызвано тем, что наличие обработчика события `onClick` подразумевает, что данный раздел `<div>` по сути функционирует как *активный* компонент. Поэтому для пользователей устройств обеспечения доступности нужно сообщить о назначении этого компонента. Средство `eslint` ожидает, что мы сделаем это, присвоив разделу `<div>` роль[4].

Мы присвоим этому разделу `<div>` роль кнопки, чтобы указать, что пользователь активизирует компонент, щелкая по нему мышью. Щелчок кнопкой мыши по панели задачи вызывает открытие всплывающего окна для редактирования этой задачи. Поэтому мы также присвоим этому разделу `<div>` атрибут `aria-haspopup`, чтобы сообщить пользователю, что щелчок по панели задачи открывает диалоговое окно. Соответствующий код приведен в листинге 9.22.

---

[4] Подробная информация о ролях и их применении приводится в *разделе 9.2*.

Листинг 9.22. Код для присвоения роли и атрибута разделу <div> компонента Task

```
<li className='Task'>
      <div className='Task-contents'
         role='button'
         aria-haspopup='dialog'
         onClick={onEdit}
      >
   ....
</div>
</li>
```

Часто вместо использования роли `button` лучше преобразовать элемент в собственно тег кнопки. Но в данном случае раздел `<div>` заключает в себя довольно крупный блок текста HTML, поэтому будет более разумным присвоить ему роль, вместо того, чтобы решать задачу, как сделать, чтобы серая кнопка выглядела как карточка.

Если исполнить `eslint` снова, то опять будут выданы два сообщения об ошибках. Но теперь одно из этих сообщений новое (листинг 9.23).

Листинг 9.23. Сообщения об ошибка после повторного исполнения eslint

```
$ npm run lint
> app@0.1.0 lint app
> eslint src
app/src/Task/Task.js
   6:9 error Visible, non-interactive elements with click handlers
             must have at least one keyboard listener
             jsx-a11y/click-events-have-key-events
             (Видимые, неинтерактивные элементы с обработчиками щелчков
             должны иметь, по крайней мере, один слушатель клавиатуры)
   6:9 error Elements with the 'button' interactive role must be tabbable
   (Элементы с интерактивной ролью "button" должны быть табулируемыми)
             jsx-a11y/interactive-supports-focus
✖ 2 problems (2 errors, 0 warnings)
```

Мы уже сказали, что панель задачи ведет себя как кнопка. Но роли должны подчиняться правилам. Если мы хотим, чтобы что-то считалось кнопкой, оно должно вести себя как кнопка. Одно из свойств кнопки — возможность перехода к ней и установки на ней фокуса посредством клавиши <Tab>. Эту возможность можно реализовать, добавив в компонент `Task` атрибут `tabIndex`, как показано в листинге 9.24.

Листинг 9.24. Добавление атрибута tabIndex в компонент Task

```
<li className='Task'>
   <div className='Task-contents'
```

```
        role='button'
        tabIndex={0}
        onClick={onEdit}
        >
    ....
    </div>
</li>
```

Значение 0 атрибута tabIndex означает, что наша задача будет частью набора табулируемых элементов страницы (т. е. элементов, между которыми можно перемещаться при помощи клавиши <Tab>).

Атрибут tabIndex может иметь несколько значений. Значение -1 означает, что соответствующий элемент может получать фокус только программными средствами, а значение 0 означает, что это обычный табулируемый компонент. Если tabIndex больше нуля, значит, система фокусирования должна дать ему более высокий приоритет. Обычно лучше избегать значений больше нуля, поскольку они могут создавать проблемы с доступностью[5].

Если исполнить eslint снова, появится всего лишь одно сообщение об ошибке (листинг 9.25).

Листинг 9.25. Сообщение об ошибке после добавления атрибута tabIndex

```
$ npm run lint
> app@0.1.0 lint app
> eslint src
app/src/Task/Task.js
   6:9 error Visible, non-interactive elements with click handlers
          must have at least one keyboard listener
          jsx-a11y/click-events-have-key-events
          (Видимые неинтерактивные элементы с обработчиками щелчков
          должны иметь, по крайней мере, один слушатель клавиатуры)
1 problems (1 errors, 0 warnings)
```

Это сообщение означает, что у нас есть событие onClick, чтобы сообщать, что происходит при щелчке мышью по панели задачи, но нет кода для его обработки. Если у пользователя нет возможности использовать клавиатуру, то он не сможет редактировать задачу.

Поэтому нам необходимо добавить какой-либо обработчик нажатия клавиши. Мы добавим код для вызова события редактирования при нажатии пользователем клавиши <Enter> или пробела (листинг 9.26).

---

[5] Проблемы, связанные со значениями tabIndex больше нуля, рассматриваются в *разделе 9.8*.

Листинг 9.26. Код для вызова события редактирования

```
<li className="Task">
  <div
    className="Task-contents"
    role="button"
    tabIndex={0}
    onClick={onEdit}
    onKeyDown={(evt) => {
      if (evt.key === 'Enter' || evt.key === ' ') {
        evt.preventDefault()
        onEdit()
      }
    }}
  >
    ....
  </div>
</li>
```

Добавление обработчика клавиатуры устраняет оставшуюся ошибку.

Для каждого из правил для модуля `jsx-a11y` есть страница на веб-сайте GitHub (**https://oreil.ly/uo7Ry**), содержащая дополнительную информацию о том, как может возникнуть нарушение данного правила и что делать, чтобы не допустить этого.

Обсуждение

Модуль `jsx-a11y`, наверное, является одним из наиболее полезных для средства `eslint`. Часто правила контроля качества кода проверяют стиль программирования и могут обнаружить значительное число проблем с ним. Но подключаемый модуль `jsx-a11y` способен по-настоящему изменить дизайн разрабатываемого приложения.

Возможность навигации в приложении посредством клавиатуры важна не только для тех, кто использует средства обеспечения доступности, но также и для тех, кто часто пользуется вашим приложением. Пользователи, постоянно работающие с приложением, часто предпочитают клавиатуру вместо мыши, поскольку это требует меньше движений и обеспечивает бóльшую точность.

Мы также рассмотрели, как посредством настроек атрибута `tabIndex` элементам страницы можно придать клавиатурную фокусировку. Некоторые браузеры (особенно Firefox) предоставляют неявные признаки наличия клавиатурного фокуса у элементов. Ясно указать пользователю, где в настоящее время находится фокус, можно, добавив в приложение следующий код CSS высокого уровня:

```
:focus-visible {
  outline: 2px solid blue;
}
```

Это правило стиля добавляет различимое очертание вокруг компонента, имеющего клавиатурный фокус. Некоторые пользователи с большей вероятностью предпочтут осуществлять навигацию посредством клавиатуры, увидев, что эта опция доступна.

Исходный код данного рецепта можно загрузить на веб-сайте GitHub по адресу **https://oreil.ly/OGfgA**.

9.4. Динамический анализ посредством axe DevTools

ЗАДАЧА

Инструменты статического анализа, такие как `eslint`, могут помочь обнаружить многие проблемы с доступностью. Но возможности статического анализа ограничены, и он часто не способен выявить ошибки времени исполнения.

При исполнении программа может функционировать таким образом, который инструменты статического анализа не в состоянии предвидеть. Нам нужно проверить доступность приложения при его исполнении в веб-браузере.

РЕШЕНИЕ

Эту задачу можно решить, установив в браузер подключаемый модуль axe DevTools. Данный модуль предоставляется как для браузера Firefox (**https://oreil.ly/S1TcB**), так и для браузера Chrome (**https://oreil.ly/MhiK0**).

После установки модуля в консоли разработчика браузера появляется дополнительная вкладка (рис. 9.8).

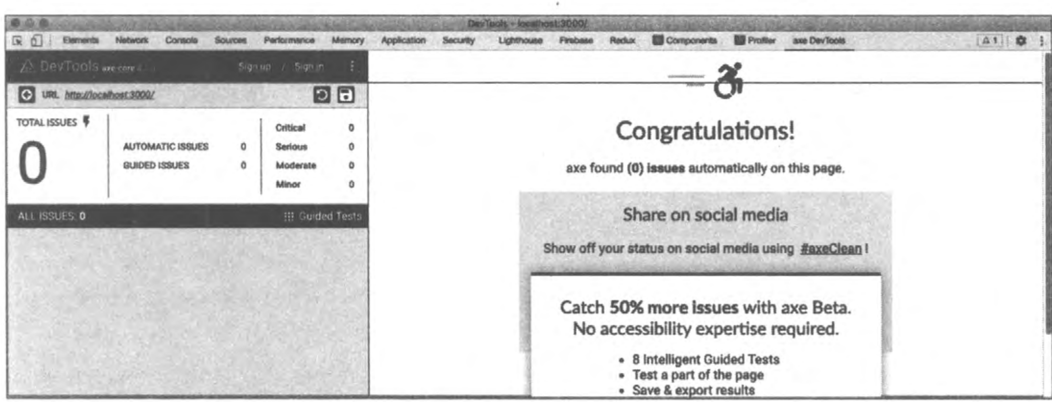


Рис. 9.8. Вкладка axe DevToll в консоли разработчика браузера

Чтобы разобраться с работой данного инструмента, приведем в негодность немного кода в примере приложения, описываемого в этой главе.

Это приложение содержит компонент `TaskForm`, который отображает всплывающее окно. Компоненту была присвоена роль `dialog`, которую мы изменим на какое-либо недействительное значение, как показано в листинге 9.27.

Листинг 9.27. Присвоение недействительной роли компоненту TaskForm

```
const TaskForm = ({ task, contexts, onCreate, onClose, open }) => {
  ...
  return (
    <Modal
      title="Create or edit a task"
      role="fish"
      open={open}
      onCancel={close}
    >
      <form>...</form>
      <ModalFooter>...</ModalFooter>
    </Modal>
  )
}
```

Откроем приложение в браузере по адресу **http://localhost:3000** и нажмем кнопку + для создания задачи. Откроется диалоговое окно создания задачи (рис. 9.9).

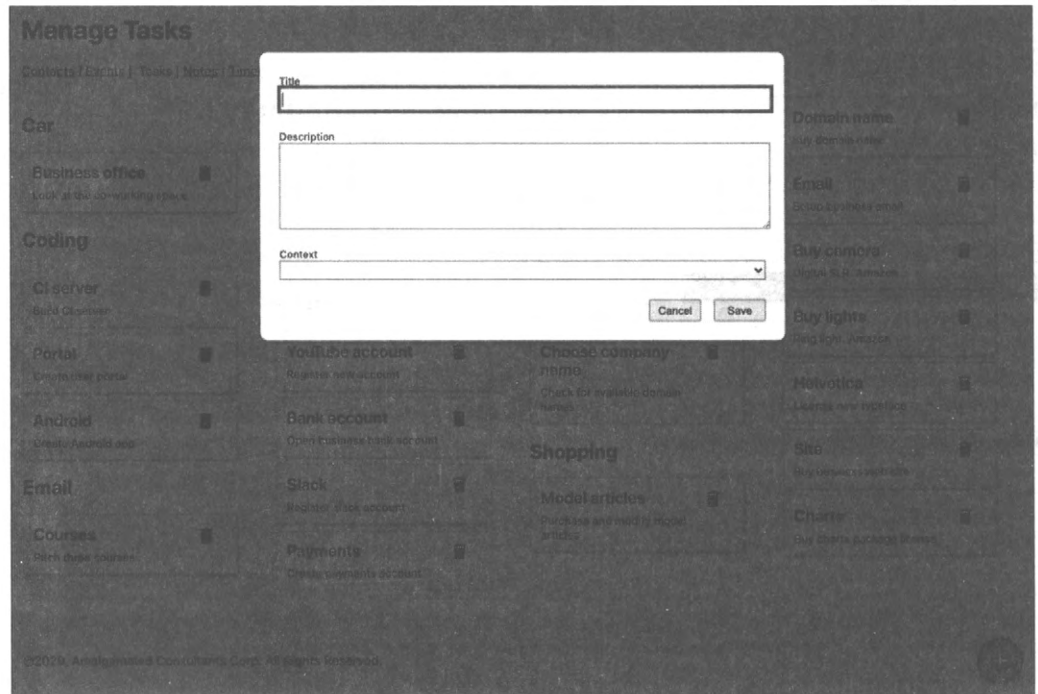


Рис. 9.9. Нажатие кнопки + открывает всплывающую форму для создания новой задачи

Теперь откроем в браузере панель инструментов разработчика, перейдем на вкладку **axe DevTools** и выполним аудит страницы. В результате будут обнаружены две ошибки (рис. 9.10).

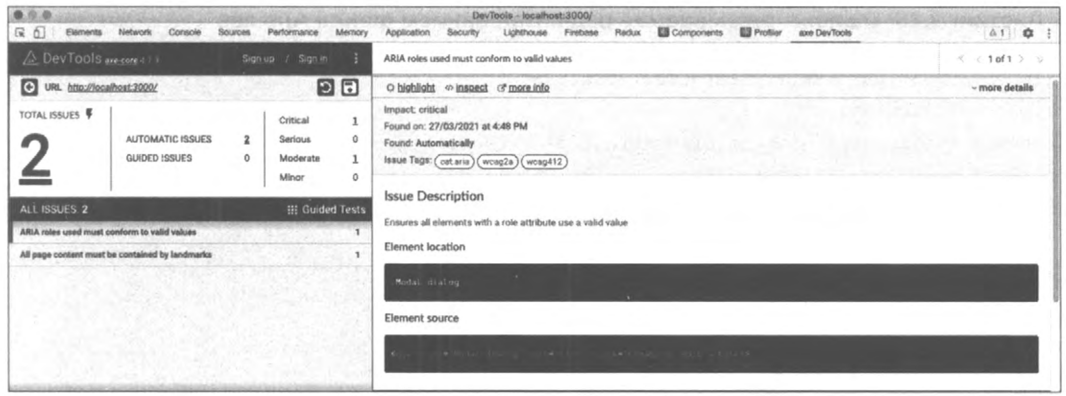


Рис. 9.10. Присвоение компоненту недействительной роли приводит к двум ошибкам

Первая ошибка вызвана тем, что компонент диалогового окна больше не имеет действительной роли, а вторая — тем, что он больше не имеет роли `dialog`, что означает, что он больше не может служить в качестве ориентира.

Некоторые роли, такие как `dialog`, помечают компонент как необходимый элемент страницы. Все части приложения должны находиться внутри ориентира.

Если выполнить сброс кода и обновить аудит axe DevTools, то сообщения об ошибках исчезнут.

Можно надеяться, что в будущем средства статического анализа кода будут поддерживать сканирование всего кода на выявление недействительных значений ролей[6]. Но средство axeDevTools может также выполнять проверку на наличие других, более тонких, проблем.

В рассматриваемом примере приложения отредактируйте файл App.css, добавив в него следующий код, чтобы изменить цвет шрифта основного заголовка:

```
h1 {
  color: #9e9e9e;
}
```

Рис. 9.11. Результат редактирования файла App.css

Результаты этого редактирования не выглядят слишком радикальными (рис. 9.11), но заставляют средство axe DevTools выдавать ошибку (листинг 9.28).

[6] К тому времени, когда вы будете читать эту книгу, такое средство, возможно, уже будет существовать.

Листинг 9.28. Ошибка, вызываемая редактированием файла App.css

```
Elements must have sufficient color contrast
Fix the following:
Element has insufficient color contrast of 2.67 (foreground color: #9e9e9e,
background color: #ffffff, font size: 24.0pt (32px), font weight: bold).
Expected contrast ratio of 3:1

(Между цветами элементов должен быть достаточный контраст)
(Исправьте следующее:)
(Контраст цвета элемента величиной 2.67 недостаточен)
(foreground color: #9e9e9e,
background color: #ffffff, font size: 24.0pt (32px), font weight: bold).
(Ожидается коэффициент контрастности величиной 3:1)
```

Ошибки с неверно заданным контрастом сравнительно легко исправить в консоли разработчика браузера Chrome. Выберите для исследования заголовок <h1>, в частности цветовой стиль этого элемента, а затем щелкните по значку панели выбора цвета. В результате откроется всплывающая панель, содержащая информацию о коэффициенте контрастности элемента, вызывающего проблему (рис. 9.12).

Рис. 9.12. Панель с информацией о цветовых свойствах, вызывающих ошибку

Открыв раздел **Contrast**, можно откорректировать цветовые характеристики, чтобы они отвечали требованиям АА и ААА стандарта доступности для контраста (рис. 9.13).

Рис. 9.13. Корректировка цвета, чтобы коэффициент контраста отвечал требованиям стандарта доступности

Браузер Chrome предлагает заменить цвет № 949494 цветом № 767676. Многие люди даже не заметят разницу, но чтение данного текста будет значительно легче для людей, восприимчивых к контрасту (рис. 9.14).

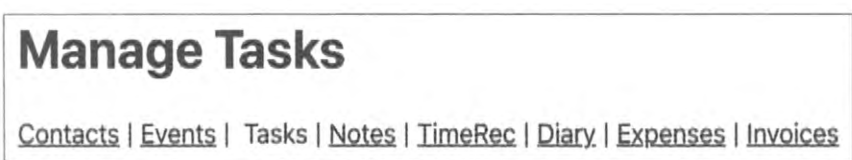


Рис. 9.14. Цвет изменен так, чтобы контраст соответствовал стандарту AAA

Иногда, если браузер Chrome не может определить конкретный цвет фона, то он не отображает информацию о контрасте. Этой проблемы можно избежать, временно присвоив проверяемому элементу атрибут `backgroundColor`.

Обсуждение

Средство axeDevTools легко использовать, но при этом оно может обнаружить многие проблемы доступности, не поддающиеся определению инструментами статического анализа.

Хотя в процессе работы разработчику приходится выполнять ручную проверку на наличие ошибок доступности, в следующей главе мы увидим, что существуют способы автоматизации тестов на доступность при помощи браузера.

Исходный код данного рецепта можно загрузить на веб-сайте GitHub по адресу **https://oreil.ly/OGfgA**.

9.5. Автоматизация тестирования в браузере посредством Cypress Axe

ЗАДАЧА

В предыдущем разделе мы выяснили, что некоторые проблемы с доступностью проявляются только при исполнении приложения в настоящем браузере, и поэтому их нельзя обнаружить при помощи инструментов статического анализа.

Если при тестировании в браузере полагаться на ручные методы, то существует большая вероятность возникновения ошибок регрессии. Было бы намного лучше автоматизировать такие типы ручных тестов, которые инструменты наподобие axe DevTools позволяют нам выполнять в браузере.

РЕШЕНИЕ

Автоматизировать тестирование на доступность в браузере можно при помощи подключаемого модуля `cypres-axe` для фреймворка тестирования Cypress. В этом модуле используется та же библиотека `axe-core`, что и в модуле axe DevTools. Тем не менее поскольку мы будем применять эту библиотеку для тестирования на уровне браузера, то сможем автоматизировать процесс проверки, чтобы сервер интеграции мог сразу же обнаруживать ошибки регрессии.

Для начала необходимо установить для нашего приложения фреймворк Cypress и библиотеку `axe-core`:

```
$ npm install --save-dev cypress axe-core
```

Затем можно установить и сам модуль `cypress-exe`:

```
$ npm install --save-dev cypress-axe
```

Если это первая установка фреймворка Cypress, то нужно запустить на исполнение ее приложение, чтобы создать соответствующие папки и начальный код, который будет служить исходным для тестов. Приложение Cypress запускается на исполнение следующей командой:

```
$ npx cypress open
```

Далее нам нужно выполнить конфигурацию модуля `cypress-exe`. Для этого отредактируем файл cypress/support/index.js, добавив в него следующую строку:

```
import 'cypress-axe'
```

Еще нам потребуется пара хуков, которые позволят записывать ошибки в процессе прогона теста. Для этого добавим в файл cypress/plugins/index.js код из листинга 9.29.

Листинг 9.29. Код хуков для добавления в файл `cypress/plugins/index.js`

```
module.exports = (on, config) => {
  on('task', {
    log(message) {
      console.log(message)
      return null
    },
    table(message) {
      console.table(message)
      return null
    },
  })
}
```

Далее удалим все примеры тестов из папки cypress/integration, создадим в ней файл accessibility.js и вставим в него код из листинга 9.30[7].

Листинг 9.30. Код файла `cypress/integraion/accessibility.js`

```
function terminalLog(violations) {
  cy.task(
    'log',
    `${violations.length} accessibility violation${
      violations.length === 1 ? '' : 's'
    } ${violations.length === 1 ? 'was' : 'were'} detected`
  )
  const violationData = violations.map(
    ({ id, impact, description, nodes }) => ({
      id,
      impact,
      description,
      nodes: nodes.length,
  })
)
  cy.task('table', violationData)
  console.table(violationData)
}
describe('can be used', () => {
  it('should be accessible when starting', () => {
    cy.visit('/')
    cy.injectAxe()
    cy.checkA11y(null, null, terminalLog)
  })
})
```

[7] Название файла не имеет значения при условии, что его расширение js и он находится в папке cypress/integration.

Этот фрагмент создан на основе кода примера из репозитория `cypres-axe` (**https://oreil.ly/2Exyx**).

Тест выполняется функцией `describe`. А функция `terminalLog` сообщает об ошибках.

Структура теста следующая:

1. Открываем страницу в корневой папке /.
2. Вставляем в страницу библиотеку `axe-core`.
3. Выполняем аудит страницы.

Библиотека `axe-core`, которая выполняет большую часть работы, также используется в других инструментах тестирования, например в модуле расширения браузера axe DevTools. Библиотека исследует текущую модель DOM, проверяя ее соответствие правилам в своей базе правил, а затем сообщает об обнаруженных ошибках.

Модуль `cypress-axe` вставляет библиотеку в `axe-core` в браузер, а затем посредством команды `checkA11y` выполняет аудит страницы. Обнаруженные проблемы отправляются функции `terminalLog`.

Если теперь исполнить наш тест в Cypress, дважды щелкнув по файлу accessibility.js, то он будет успешно выполнен, как показано на рис. 9.15.

Что ж, давайте тогда создадим проблему, добавив еще один тест (листинг 9.31).

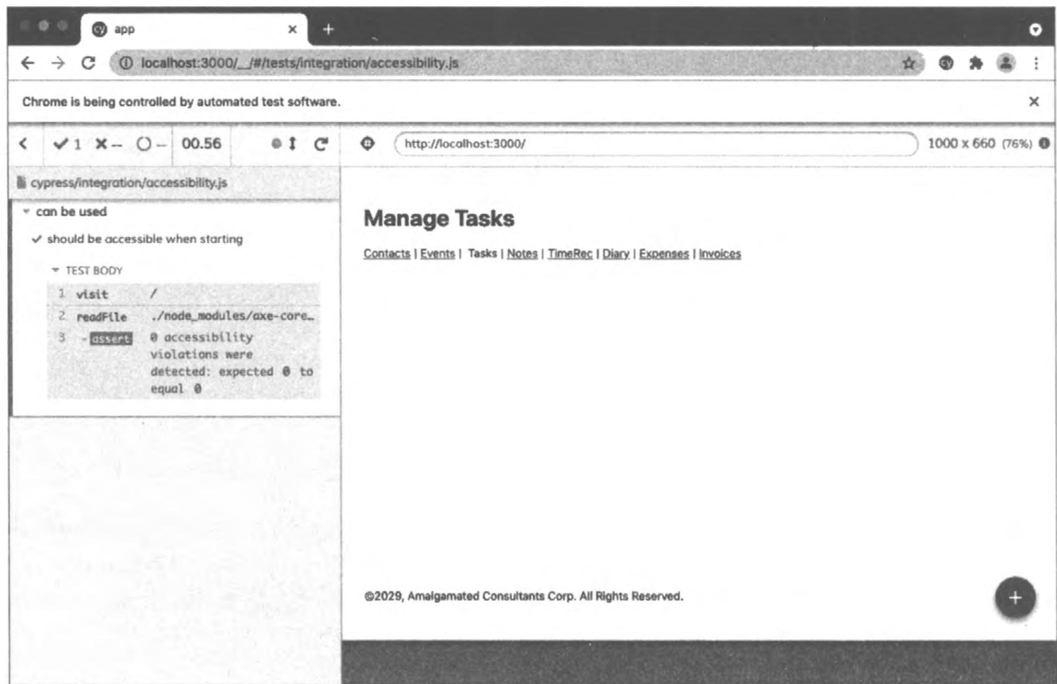


Рис. 9.15. Код страницы успешно проходит тест на доступность

Листинг 9.31. Код второго теста

```
it('should be accessible when creating a task', () => {
  cy.visit('/')
  cy.injectAxe()
  cy.contains('+').click()
  cy.checkA11y(null, null, terminalLog)
})
```

Этот тест запускает приложение, вызывает нажатие кнопки +, открывая форму для создания новой задачи, а затем выполняет проверку доступности.

В ее текущей форме приложение успешно пройдет и этот тест. Поэтому модифицируем компонент `TaskForm`, присвоив ему недействительное значение роли, как показано в листинге 9.32.

Листинг 9.32. Присвоение недействительного значения роли компоненту `TaskForm`

```
const TaskForm = ({ task, contexts, onCreate, onClose, open }) => {
  ...
  return (
    <Modal
      title="Create or edit a task"
      role="hatstand"
      open={open}
      onCancel={close}
    >
      <form>...</form>
      <ModalFooter>...</ModalFooter>
    </Modal>
  )
}
```

Если выполнить наш тест теперь, то он завершится неуспешно. Тест нужно исполнять с открытой консолью JavaScript (рис. 9.16), чтобы увидеть ошибку в таблице консоли.

Обсуждение

Замечательное введение в тему аудита доступности и тестирования при помощи модуля `cypress-axe` было представлено в докладе Марси Суттон (Marcy Sutton, **https://oreil.ly/nS6R2**) на конференции ReactJS Girls Conference. Из доклада мы впервые узнали об этом модуле и с тех пор постоянно используем его.

Исходный код данного рецепта можно загрузить на веб-сайте GitHub по адресу **https://oreil.ly/OGfgA**.

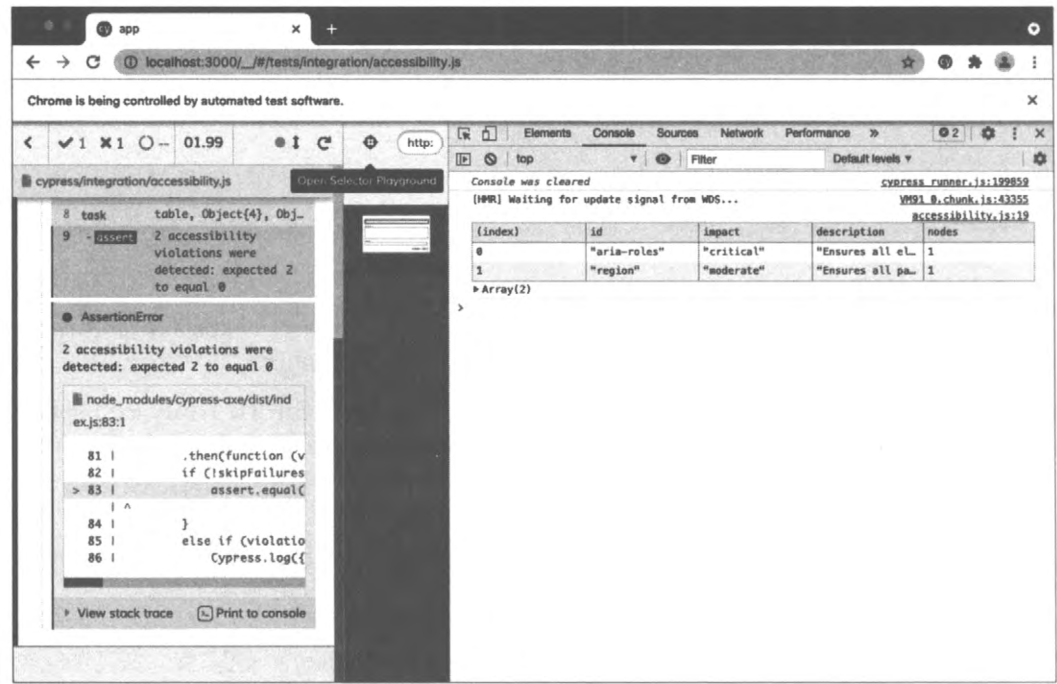


Рис. 9.16. Подробности ошибки в консоли JavaScript

9.6. Добавление в страницу кнопок пропуска содержимого

ЗАДАЧА

Часто страницы содержат большой объем информации в самом начале. Там могут быть навигационные ссылки, меню быстрых действий, ссылки на сайты социальных сетей, поля поиска и т. п. Для тех, кто знакомится с содержимым страницы визуально и пользуется мышью для навигации по ней, это не представляет какой-либо большой проблемы. Скорее всего, такой пользователь мысленно отфильтрует все это и перейдет к основному содержимому страницы.

Но тем, кто "просматривает" страницу при помощи средства чтения с экрана, может, придется прослушивать подробности всех этих начальных элементов на каждой посещаемой странице. Современная технология чтения с экрана часто позволяет пользователям автоматически переходить между разделами и заголовками, но тем не менее, чтобы вычислить, где начинается важное содержимое, может потребоваться некоторое время.

Поэтому код многих веб-сайтов содержит скрытые ссылки и кнопку с надписью типа "Перейти к содержимому", позволяющую пользователям с помощью клавиатуры перейти к началу основной части страницы.

Один из таких веб-сайтов — YouTube. Если на этом сайте нажать клавишу <Tab> три раза подряд, то в его левом верхнем углу отобразится кнопка **SKIP NAVIGATION** (рис. 9.17), после чего нажатие клавиши пробела переместит фокус клавиатуры на основное содержимое.

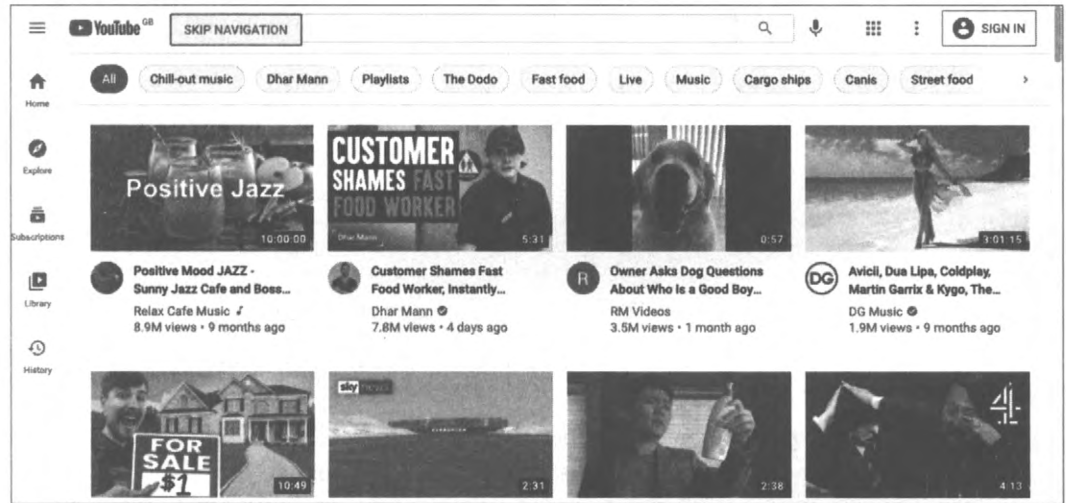


Рис. 9.17. Кнопка **SKIP NAVIGATON** на веб-сайте YouTube

Как можно создать такую кнопку, которая отображается только при навигации посредством клавиши <Tab>?

РЕШЕНИЕ

Данный рецепт содержит компонент многократного использования `SkipButton`, который можно вставлять по сути на любую страницу, не нарушая ее дизайн или разметку.

Этот компонент должен обладать следующими возможностями:

- Быть скрытым, пока не будет открыт посредством клавиши <Tab>. Просто прозрачная кнопка здесь не подойдет, поскольку пользователь может случайно нажать ее щелчком по этой части экрана.
- Отображаться поверх содержимого страницы, чтобы не оставлять для нее места в разметке.
- Функционировать как распознаваемая кнопка. Это означает, что компонент должен быть распознаваем средствами чтения с экрана и вести себя как кнопка. Когда на нем установлен фокус, он должен срабатывать при нажатии клавиши <Enter> или клавиши пробела.
- Скрываться после срабатывания.

В процессе разработки мы добавим еще несколько других требований, но перечисленных достаточно, чтобы приступить к работе.

Начнем с создания нового компонента, называющегося SkipButton. Этот компонент будет возвращать один раздел <div>, допуская включение в него любых переданных ему потомков. Соответствующий код приведен в листинге 9.33.

Листинг 9.33. Код компонента SkipButton

```
const SkipButton = (props) => {
   const { className, children, ...others } = props
   return (
      <div className={`SkipButton ${className || ''}`} {...others}>
         {children}
      </div>
   )
}
```

Компонент также будет принимать имя класса и любые другие свойства, которые могут передаваться ему родителем.

Компонент должен распознаваться средствами чтения с экрана как настоящая кнопка. Этого можно добиться, заменив раздел <div> настоящей кнопкой, но мы не будем так делать, чтобы облегчить задачу применения к нему стилистического форматирования. Но мы присвоим ему роль кнопки и, поскольку роли имеют правила, также присвоим его атрибуту tabIndex значение 0. Последнее нужно сделать в любом случае, поскольку пользователь должен быть в состоянии перейти к этой кнопке посредством клавиши <Tab>. Соответствующий код приведен в листинге 9.34.

Листинг 9.34. Присвоение компоненту SkipButton роли button

```
const SkipButton = (props) => {
   const { className, children, ...others } = props
   return (
      <div
         className={`SkipButton ${className || ''}`}
         role="button"
         tabIndex={0}
         {...others}
      >
         {children}
      </div>
   )
}
```

Нажатие кнопки должно вызывать какое-то действие. Или, вернее, действие должно вызываться нажатием клавиши <Enter> или клавиши пробела. Поэтому мы позволим данному компоненту принимать свойство onClick, но присоединим к нему

обработчик события, срабатывающий при нажатии клавиши <Enter> или клавиши пробела. Соответствующий код приведен в листинге 9.35.

Листинг 9.35. Добавление в компонент `SkipButton` свойства `onClick`

```
const SkipButton = (props) => {
   const { className, children, onClick, ...others } = props
   return (
      <div
         className={`SkipButton ${className || ''}`}
         role="button"
         tabIndex={0}
         {...others}
         onKeyDown={(evt) => {
            if (evt.key === 'Enter' || evt.key === ' ') {
               evt.preventDefault()
               onClick(evt)
            }
         }}
      >
         {children}
      </div>
   )
}
```

Конечно же, мы могли бы назвать это свойство `onKeyDown`, но кнопки обычно имеют свойства `onClick`, и это название легче запомнить, когда мы будем использовать его.

Наконец, мы позволим компоненту принимать атрибут `ref`, что будет полезным в следующем рецепте.

Атрибуты `ref` нельзя передавать таким же образом, как большинство других свойств. Отрисовщик React с помощью атрибута `ref` отслеживает в модели DOM сгенерированные элементы.

Чтобы компонент мог принимать объект `ref`, нам нужно заключить весь компонент в оболочку в вызове функции React `forwardRef`. Эта функция возвращает заключенную в оболочку версию компонента, извлекая атрибут `ref` из родительского компонента и явно передавая его облекаемому ею компоненту. Это может звучать немного сложновато, но в реальности означает лишь то, что показано в листинге 9.36.

Листинг 9.36. Полный код компонента `SkipButton` со всеми модификациями

```
import { forwardRef } from 'react'
import './SkipButton.css'
const SkipButton = forwardRef((props, ref) => {
   const { className, children, onClick, ...others } = props
```

```
  return (
    <div
      className={`SkipButton ${className || ''}`}
      role="button"
      tabIndex={0}
      ref={ref}
      {...others}
      onKeyDown={(evt) => {
        if (evt.key === 'Enter' || evt.key === ' ') {
          evt.preventDefault()
          onClick(evt)
        }
      }}
    >
      {children}
    </div>
  )
})
```

В листинге 9.36 приведен полный код компонента `SkipButton`, включая импортирование информации о стилевом оформлении. Это всего лишь кнопка. Все остальное осуществляется оформлением стиля в файле SkipButton.css.

Нам нужно, чтобы наша кнопка отображалась поверх всего остального содержимого страницы, поэтому мы присвоим ее атрибуту `z-index` какое-либо по-настоящему большое значение:

```
.SkipButton {
  z-index: 10000;
}
```

Кнопка должна быть скрыта до тех пор, пока пользователь не перейдет к ней посредством клавиши <Tab>. Это можно было бы реализовать, сделав кнопку прозрачной, но появились бы две проблемы. Первая состоит в том, что кнопка может разместиться поверх какого-либо активируемого щелчком элемента. В таком случае он заблокирует щелчки, если только атрибутам `pointer-events` не присвоить значение `none`. Другая проблема заключается в том, что кнопка, хотя и прозрачная, все-таки находится на экране, и может оказаться еще одной помехой для средства чтения с экрана. Если средство считывания с экрана пространственно преобразует экран в код Брайля, то пользователь может услышать "Перейти к содержимому" посередине фрагмента текста.

Поэтому мы не будем размещать нашу кнопку на экране до тех пор, пока она нам не понадобится. Соответствующий код CSS приведен в листинге 9.37.

Листинг 9.37. Код CSS для размещения кнопки вне экрана

```
.SkipButton {
  z-index: 10000;
  position: absolute;
```

```
   left: -1000px;
   top: -1000px;
}
```

Что же происходит, когда кнопка получает фокус? Мы можем задать стили, которые применяются только в этом случае. Соответствующий код CSS приведен в листинге 9.38.

Листинг 9.38. Код CSS для применения к кнопке при получении фокуса

```
.SkipButton {
   z-index: 10000;
   position: absolute;
   left: -1000px;
   top: -1000px;
}
.SkipButton:focus {
   top: auto;
   left: auto;
}
```

Помимо этого, можно просто добавить немного визуального стилистического оформления (листинг 9.39). Важно помнить, что не все, кому придется иметь дело с этой кнопкой, будут работать со средством считывания с экрана. Некоторые пользователи могут применять клавиатурную навигацию потому, что они не могут задействовать мышь или просто считают, что для них это удобнее.

Листинг 9.39. Дополнительное стилистическое оформление кнопки

```
.SkipButton {
   z-index: 10000;
   position: absolute;
   left: -1000px;
   top: -1000px;
   font-size: 12px;
   line-height: 16px;
   display: inline-block;
   color: black;
   font-family: sans-serif;
   background-color: #ffff88;
   padding: 8px;
   margin-left: 8px;
}

.SkipButton:focus {
   top: auto;
   left: auto;
}
```

Теперь мы можем вставить компонент `SkipButton` где-либо возле начала страницы. Хотя он не будет видимым до тех пор, пока не получит фокус, его позиция на экране имеет значение. Он должен быть на расстоянии двух или трех нажатий клавиши <Tab> от начала страницы. Мы добавим его в раздел `<header>`, как показано в листинге 9.40.

Листинг 9.40. Размещение компонента `SkipButton` на странице

```
<header>
  <SkipButton onClick={() =>
  document.querySelector('.addButton').focus()}>
    Skip to content
  </SkipButton>
  <h1>Manage Tasks</h1>
</header>
```

Здесь мы находим элемент, который получит фокус, просто используя метод `document.querySelector`. Элемент, который нужно пропустить, можно было бы снабдить ссылкой, или иначе перейти к требуемому элементу. Но мы открыли для себя на практике, что наиболее прямолинейным подходом будет вызов простого метода `document.querySelector`. Он позволяет с легкостью обращаться к элементам, которые могут отсутствовать в текущем компоненте. Он также не полагается на переход к закладке в странице, что может оказаться неработоспособным в случае изменения метода маршрутизации приложения.

Если мы теперь откроем приложение в браузере и нажмем клавишу <Tab>, то в левом верхнем углу приложения должен отобразиться компонент `SkipButton` в виде кнопки **Skip to content** (рис. 9.18).

Обсуждение

Желательно разместить компонент `SkipButton` на расстоянии трех нажатий клавиши <Tab> от начала страницы. Также полезно сделать количество нажатий клавиши <Tab> для перехода к этой кнопке одинаковым для всех страниц приложения. Тогда пользователь вскоре вычислит, как перескакивать к важной части каждой страницы. Мы также обнаружили, что такие кнопки пропуска второстепенного содержимого пользуются популярностью у людей, находящих работу с клавиатурой более продуктивной, нежели с мышью.

Можно также создать стандартную кнопку пропуска содержимого для каждой страницы, которая перемещает фокус на первый табулируемый элемент главного раздела `<main>` страницы[8].

Исходный код данного рецепта можно загрузить на веб-сайте GitHub по адресу **https://oreil.ly/0GfgA**.

[8] Дополнительная информация о разделах `<main>` приводится в *разделе 9.1*.

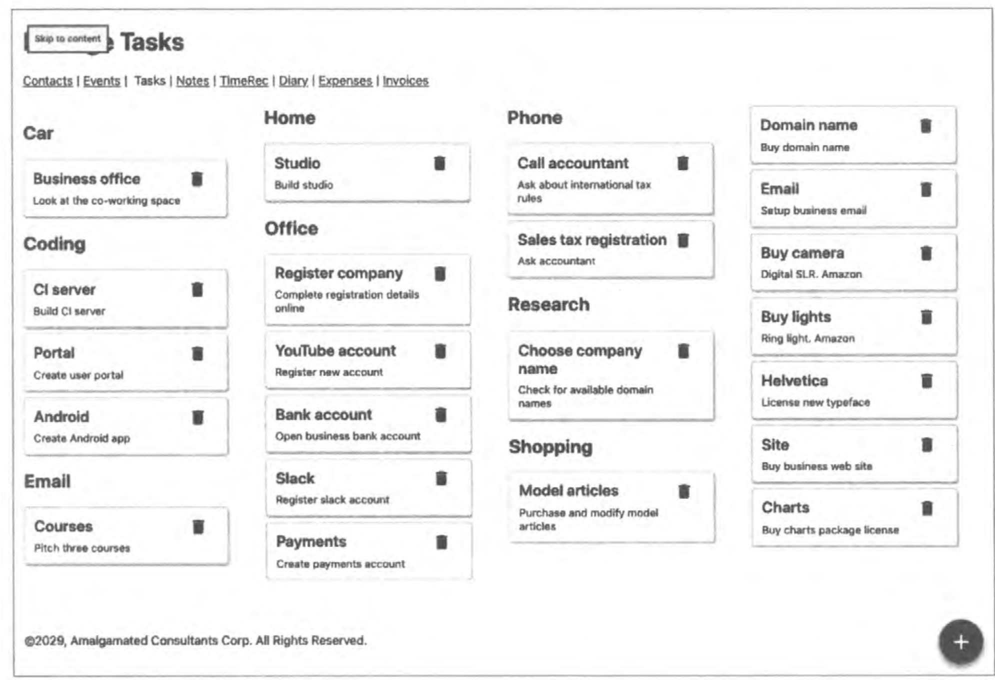


Рис. 9.18. Отображение кнопки перехода к содержимому при нажатии клавиши <Tab>

9.7. Добавление возможности пропуска областей страницы

ЗАДАЧА

В *разделе 9.6* мы увидели, что кнопки для пропуска содержимого позволяют пользователю быстро пройти мимо второстепенных объектов в виде заголовков и элементов навигации в начале страницы и добраться до основного содержимого.

Но возможны ситуации, когда пользователь хотел бы иметь возможность пропустить набор компонентов даже в основном содержимом. Рассмотрим пример приложения создания и редактирования задач, которое мы рассматриваем в этой главе. Пользователь может создать довольно большое количество задач в каждой группе (рис. 9.19).

В данном случае если пользователь хочет перейти к группе задач `Shopping`, то ему для этого пришлось бы пройти через все другие 14 предшествующих задач. Более того, каждая из этих задач имеет две фокусные точки: сама задача и кнопка удаления этой задачи. Следовательно, даже после перехода в начало основного контента, чтобы добраться до требуемого содержимого, пользователю придется пройти через каждую из 28 фокусных точек.

Как можно предоставить пользователю возможность пропустить набор компонентов?

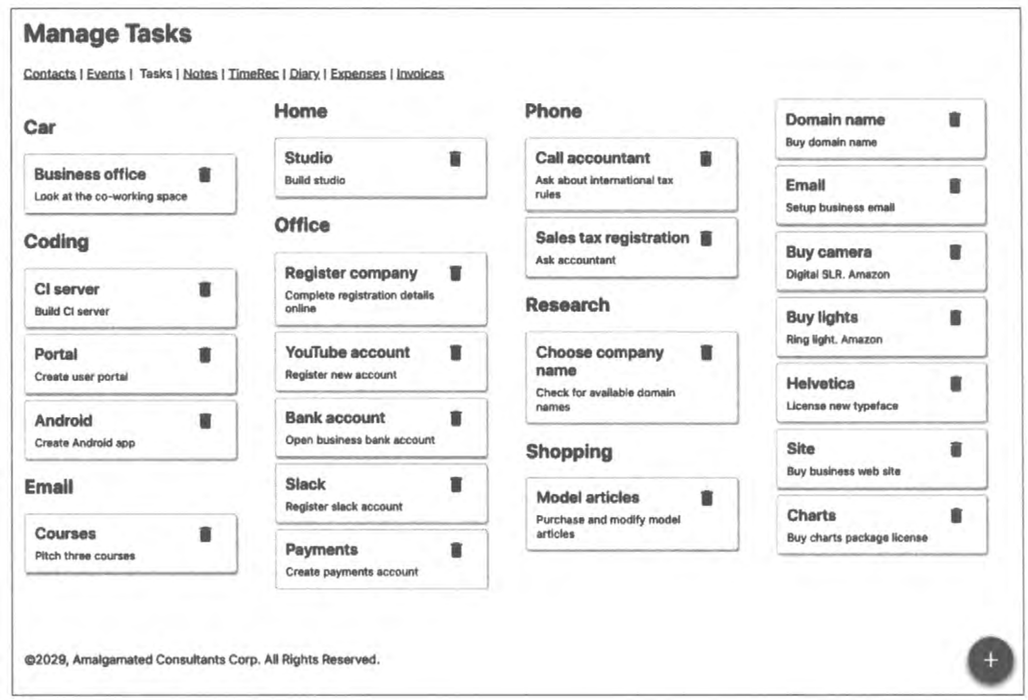


Рис. 9.19. Наборы задач, упорядоченные по группам

РЕШЕНИЕ

Данную задачу можно решить, создав пропускаемые области (т. е. набор компонентов) при помощи созданного в предыдущем разделе компонента `SkipButton`.

Например, перемещаясь из начала в конец при переходе в какой-либо раздел основного содержимого страницы, например раздел задач **Office**, нужно отобразить кнопку, позволяющую пользователю полностью пропустить все задачи этого раздела и перейти в следующий раздел (рис. 9.20).

Подобным образом мы хотим, чтобы эта кнопка также отображалась и при переходе в раздел **Office**, перемещаясь с конца в начало содержимого, также позволяя пропустить все задачи этого раздела, но теперь перейти в предыдущий раздел (рис. 9.21).

Эти кнопки должны отображаться только при входе в группу, но не при выходе из нее. Это означает, что кнопка **Skip Office** отображается только при перемещении вперед посредством нажатия клавиши <Tab>, а кнопка **Skip before Office** — только при перемещении назад.

Прежде чем вдаваться во все тонкости реализации возможности пропуска области, рассмотрим, как мы будем использовать эту возможность. Наш пример приложения отрисовывает ряд групп задач с помощью компонента `TasksContexts`, как показано в листинге 9.41.

Manage Tasks

Contacts | Events | Tasks | Notes | TimeRec | Diary | Expenses | Invoices

Car

Business office
Look at the co-working space

Coding

CI server
Build CI server

Portal
Create user portal

Android
Create Android app

Email

Courses
Pitch three courses

Home

Studio
Build studio

Skip Office

Register company
Complete registration details online

YouTube account
Register new account

Bank account
Open business bank account

Slack
Register slack account

Payments
Create payments account

Phone

Call accountant
Ask about international tax rules

Sales tax registration
Ask accountant

Research

Choose company name
Check for available domain names

Shopping

Model articles
Purchase and modify model articles

Domain name
Buy domain name

Email
Setup business email

Buy camera
Digital SLR. Amazon

Buy lights
Ring light. Amazon

Helvetica
License new typeface

Site
Buy business web site

Charts
Buy charts package license

©2029, Amalgamated Consultants Corp. All Rights Reserved.

+

Рис. 9.20. Кнопка пропуска группы для перехода в следующую группу

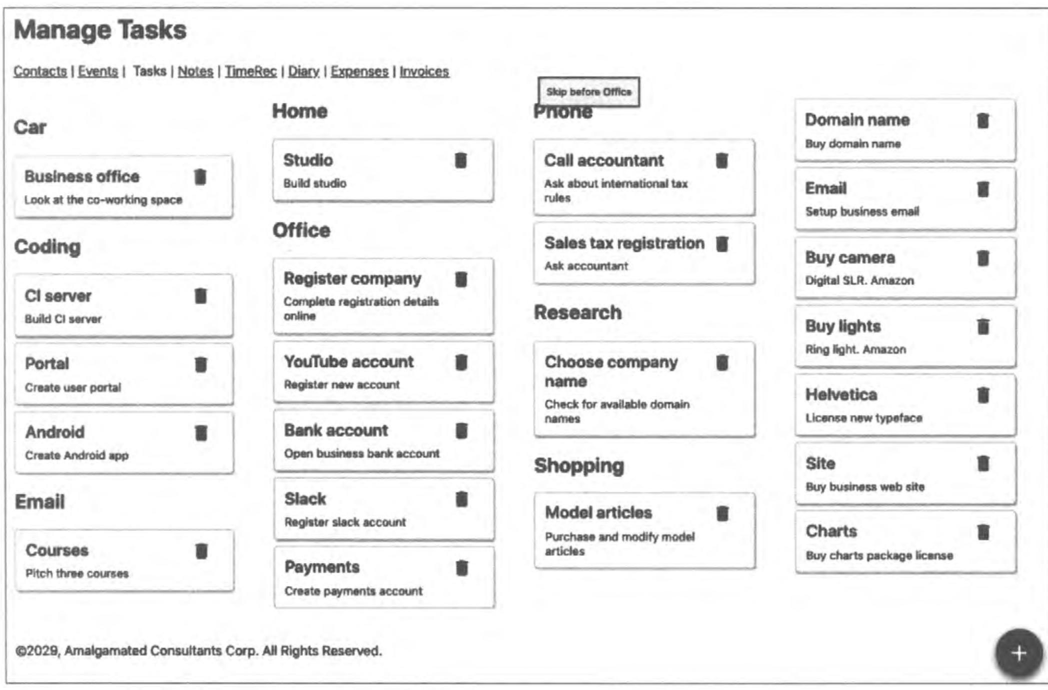


Рис. 9.21. Кнопка пропуска группы для перехода в предыдущую группу

Листинг 9.41. Отрисовка ряда групп задач посредством компонента TasksContents

```
import TaskList from '../TaskList'
import './TaskContexts.css'
function TaskContexts({ contexts, tasks, onDelete, onEdit }) {
   return contexts.map((c) => {
      const tasksForContext = tasks.filter((t) => t.context === c.value)
      if (tasksForContext.length === 0) {
         return <div className="TaskContexts-context"> </div>
      }
      return (
         <div key={c.value} className="TaskContexts-context">
            <h2>{c.name}</h2>
            <TaskList
               tasks={tasksForContext}
               onDelete={onDelete}
               onEdit={onEdit}
            />
         </div>
      )
   })
}
export default TaskContexts
```

Каждый "context" (т. е. группа задач для покупок Shopping, офиса Office, исследований Research и т. д.) имеет заголовок и список задач. Мы хотим, чтобы у пользователя была возможность полностью пропустить любую из этих групп при перемещении как вперед, так и назад. Для этого мы сначала поместим каждую группу задач в оболочку нового компонента `Skip`, как показано в листинге 9.42.

Листинг 9.42. Помещение группы задач в оболочку компонента Skip

```
import TaskList from '../TaskList'
import Skip from '../Skip'
import './TaskContexts.css'
function TaskContexts({ contexts, tasks, onDelete, onEdit }) {
   return contexts.map((c) => {
      const tasksForContext = tasks.filter((t) => t.context === c.value)
      if (tasksForContext.length === 0) {
         return <div className="TaskContexts-context"> </div>
      }
      return (
         <div key={c.value} className="TaskContexts-context">
            <Skip name={c.name}>
               <h2>{c.name}</h2>
               <TaskList
                  tasks={tasksForContext}
```

```
                onDelete={onDelete}
                onEdit={onEdit}
              />
            </Skip>
          </div>
        )
    })
}
export default TaskContexts
```

В результате такого размещения группы задач в (пока несуществующем) компоненте `Skip` при каждом входе в группу задач будет отображаться компонент `SkipButtons`.

Компоненту `Skip` нужно передавать только имя группы, которое будет отображаться в надписи кнопки "Skip..." и "Skip before...".

Чтобы создать компонент `Skip`, начнем с простого компонента, который отрисовывает два компонента `SkipButton` и любой переданный им дочерний компонент (листинг 9.43).

Листинг 9.43. Код для отрисовки компонентов `SkipButton`

```
import { useRef } from 'react'
import SkipButton from '../SkipButton'
import './Skip.css'
const Skip = ({ children, name }) => {
   const startButton = useRef()
   const endButton = useRef()
return (
   <div className="Skip">
      <SkipButton ref={startButton}>Skip {name}</SkipButton>
      {children}
      <SkipButton ref={endButton}>Skip before {name}</SkipButton>
      </div>
   )
}
```

Здесь мы создали два объекта `ref`, которые позволят нам отслеживать каждую кнопку. Нажатие пользователем кнопки `startButton` будет устанавливать фокус на кнопке `endButton`, и наоборот, как показано в листинге 9.44.

Листинг 9.44. Установка фокуса на кнопках `startButton` и `endButton`

```
import { useRef, useState } from 'react'
import SkipButton from '../SkipButton'
import './Skip.css'
```

```
const Skip = ({ children, name }) => {
  const startButton = useRef()
  const endButton = useRef()
  const skipAfter = () => {
    if (endButton.current) {
      endButton.current.focus()
    }
  }
  const skipBefore = () => {
    if (startButton.current) {
      startButton.current.focus()
    }
  }
  return (
    <div className="Skip">
      <SkipButton ref={startButton} onClick={skipAfter}>
        Skip {name}
      </SkipButton>
      {children}
      <SkipButton ref={endButton} onClick={skipBefore}>
        Skip before {name}
      </SkipButton>
    </div>
  )
}
```

В результате исполнения этого кода при входе в группу задач будет отображаться кнопка `startButton` в начале группы. Последующее нажатие клавиши <Enter> установит фокус на кнопке `endButton` в конце списка задач группы.

Но вместо перехода к кнопке `endButton` мы хотим установить фокус на следующем после этой кнопки элементе. Иными словами, мы как будто хотим перейти к кнопке в конце списка, а затем сразу же нажать клавишу <Tab>, чтобы перейти к следующему элементу. Это можно сделать, создав функцию `focusNextElement`, которая будет программно выполнять действие нажатия клавиши <Tab>[9].

Листинг 9.45. Функция для программного нажатия клавиши <Tab>

```
const focusableSelector = 'a[href], ..., *[contenteditable]'
function focusNextElement() {
  var focusables = document.querySelectorAll(focusableSelector)
  var current = document.querySelectorAll(':focus')
  var nextIndex = 0
```

[9] Этот подход основан на ответе пользователя Радека (**https://oreil.ly/5p8nS**) на вопрос на веб-сайте StackOverflow (**https://oreil.ly/Li5sB**).

```
  if (current.length === 1) {
    var currentIndex = Array.prototype.indexOf.call(
      focusables,
      current[0]
    )
    if (currentIndex + 1 < focusables.length) {
      nextIndex = currentIndex + 1
    }
  }
  focusables[nextIndex].focus()
}
```

Этот код сначала обнаруживает все элементы в модели DOM, к которым можно переходить нажатием клавиши <Tab>. Затем он находит в этом списке элемент, на котором в данный момент установлен фокус, а затем устанавливает фокус на следующий за ним элемент.

Подобную функцию `focusPreviousElement` можно создать и для программного обратного нажатия клавиши <Tab> для установки фокуса на элементе, предшествующем текущей группе. Затем можно добавить наш компонент `Skip`. Соответствующий код приведен в листинге 9.46.

Листинг 9.46. Добавление функций для программного нажатия клавиши <Tab>

```
import { useRef, useState } from 'react'
import {
  focusNextElement,
  focusPreviousElement,
} from './focusNextElement'
import SkipButton from '../SkipButton'
import './Skip.css'
const Skip = ({ children, name }) => {
  const startButton = useRef()
  const endButton = useRef()
  const skipAfter = () => {
    if (endButton.current) {
      endButton.current.focus()
      focusNextElement()
    }
  }
  const skipBefore = () => {
    if (startButton.current) {
      startButton.current.focus()
      focusPreviousElement()
    }
  }
```

```
   return (
      <div className="Skip">
         <SkipButton ref={startButton} onClick={skipAfter}>
            Skip {name}
         </SkipButton>
         {children}
         <SkipButton ref={endButton} onClick={skipBefore}>
            Skip before {name}
         </SkipButton>
      </div>
   )
}
```

При переходе в группу задач (например, Office) отображается кнопка пропуска, которая позволит нам полностью пропустить группу, переместившись на следующий элемент.

Нам осталось добавить еще одну возможность. Кнопки пропуска группы должны отображаться только при входе в группу, но не при выходе из нее. Этого можно добиться, обновляя переменную `inside` значением, соответствующим присутствию или отсутствию фокуса на текущем компоненте, как показано в листинге 9.47.

Листинг 9.47. Использование переменной `inside`

```
import { useRef, useState } from 'react'
import {
   focusNextElement,
   focusPreviousElement,
} from './focusNextElement'
import SkipButton from '../SkipButton'
import './Skip.css'
const Skip = ({ children, name }) => {
   const startButton = useRef()
   const endButton = useRef()
   const [inside, setInside] = useState(false)
   const skipAfter = () => {
      if (endButton.current) {
         endButton.current.focus()
         focusNextElement()
      }
   }
   const skipBefore = () => {
      if (startButton.current) {
         startButton.current.focus()
         focusPreviousElement()
      }
   }
```

```
  return (
    <div
      className="Skip"
      onFocus={(evt) => {
        if (
          evt.target !== startButton.current &&
          evt.target !== endButton.current
        ) {
          setInside(true)
        }
      }}
      onBlur={(evt) => {
        if (
          evt.target !== startButton.current &&
          evt.target !== endButton.current
        ) {
          setInside(false)
        }
      }}
    >
      <SkipButton
        ref={startButton}
        tabIndex={inside ? -1 : 0}
        onClick={skipAfter}
      >
        Skip {name}
      </SkipButton>
      {children}
      <SkipButton
        ref={endButton}
        tabIndex={inside ? -1 : 0}
        onClick={skipBefore}
      >
        Skip before {name}
      </SkipButton>
    </div>
  )
}
```

Теперь наш компонент для пропуска области полностью готов. При перемещении пользователя посредством клавиши <Tab> в группу задач отображается кнопка пропуска области `SkipButton`, предоставляющая пользователю возможность пропустить все элементы данной группы и сразу же перейти в следующую (или предыдущую).

Обсуждение

Не следуют злоупотреблять возможностью пропуска области. Она лучше всего подходит для пропуска нескольких компонентов, через каждый из которых в противном случае пользователю пришлось бы перемещаться нажатием клавиши <Tab> соответствующее число раз.

Но можно применить и другие подходы. Предположим, например, что страница содержит последовательность заголовков и подзаголовков. В таком случае можно добавить кнопки пропуска, позволяющие пользователю переходить на следующий (или предыдущий) заголовок.

Некоторые пользователи могут работать с ПО, обеспечивающим специальные возможности доступа, позволяющим пропускать группы и разделы компонентов, не задействуя никакого дополнительного кода в приложениях. В таких случаях кнопки пропуска не будут отображаться на странице и пользователь будет полностью их игнорировать.

Исходный код данного рецепта можно загрузить на веб-сайте GitHub по адресу **https://oreil.ly/0GfgA**.

9.8. Захват области действия в модальных окнах

ЗАДАЧА

Приложения React часто открывают всплывающие окна. Например, щелчок по панели задачи в рассматриваемом в данной главе приложении для создания и редактирования заданий открывает всплывающее диалоговое окно, в котором пользователь может ввести подробную информацию для новой задачи (рис. 9.22).

Подобные всплывающие окна часто являются модальными, и для того, чтобы возвратиться обратно в приложение, такое окно нужно закрыть. Проблема с пользовательскими модальными окнами состоит в том, что они могут терять фокус.

Рассмотрим всплывающее окно для создания новой задачи нашего примера приложения. Ранняя версия кода приложения как раз приводила к проблеме утраты фокуса. Щелчок мышью по панели задачи открывал всплывающее модальное окно-форму для редактирования данной задачи, и фокус немедленно захватывался первым полем этой формы. Но при нажатии клавиш <Shift>+<Tab> (обратная табуляция) фокус перемещался на какой-либо элемент в главном окне приложения (рис. 9.23).

Если пользователь в состоянии увидеть, что фокус ушел с формы, то он поймет, что это всего лишь немного странная особенность приложения. Но это может сильно сбивать с толку людей, работающих с ПО обеспечения доступа, которые могут ничего не знать о том, что модальное окно все еще открыто. Впечатление может быть еще более странным для людей, которые в состоянии видеть, но не могут пользоваться мышью. Они могут установить фокус на компоненте, скрытым за диалоговым окном.

Рис. 9.22. Всплывающая форма для редактирования задачи

Рис. 9.23. Переход фокуса с модальной формы на панель задачи **Charts** в главном окне приложения при нажатии клавиш <Shift>+<Tab>

Поэтому нам нужен какой-либо способ удержания фокуса в пределах набора компонентов, чтобы пользователь не мог случайно переместить его на компоненты, которые по идее должны быть недоступными.

РЕШЕНИЕ

Для решения этой задачи мы установим библиотеку Focus Lock для React, которая удерживает фокус в небольшом подмножестве компонентов. Библиотека устанавливается следующей командой:

```
$ npm install react-focus-lock
```

Библиотека Focus Lock заключает набор компонентов в оболочку компонента `ReactFocusLock`, который будет следить за фокусом, ожидая, когда он уйдет с него. Когда это случится, то компонент сразу же возвратит фокус обратно внутрь себя.

Модальное окно в нашем примере приложения создается компонентом `Modal` (листинг 9.48).

Листинг 9.48. Компонент `Modal` для создания модального окна

```
import './Modal.css'
function Modal({ open, onCancel, children, role, title }) {
   if (!open) {
      return null
   }
   return (
      <div role="presentation" className="Modal" ...>
         <div className="Modal-dialog" role={role} title={title} ...>
            {children}
         </div>
      </div>
   )
}
```

Все содержимое модального окна передается как компоненты-потомки. Мы можем удерживать фокус в пределе набора этих компонентов-потомков, заключив их в оболочку компонента `ReactFocusLock`, как показано в листинге 9.49.

Листинг 9.49. Модальное окно в оболочке `ReactFocusLock`

```
import ReactFocusLock from 'react-focus-lock'
import './Modal.css'
function Modal({ open, onCancel, children, role, title }) {
   if (!open) {
      return null
   }
```

```
  return (
    <div role="presentation" className="Modal" ...>
      <div className="Modal-dialog" role={role} title={title} ...>
        <ReactFocusLock>{children}</ReactFocusLock>
      </div>
    </div>
  )
}
```

Теперь при многократном нажатии клавиши <Tab> в открытом модальном окне `TaskForm` фокус будет последовательно перемещаться по кнопкам и полям этого диалогового окна. При нажатии клавиши <Tab>, когда фокус находится на последнем элементе окна, он перейдет на первый элемент окна, а при нажатии клавиш <Shift>+<Tab>, когда фокус находится на первом элементе, фокус переместится на последний элемент.

Библиотека Focus Lock работает, создавая скрытую кнопку, атрибуту которой `tabIndex` присваивается значение `1`. Это нарушает правило `tabindex` библиотеки `axe-core`, гласящее, что значение этого атрибута не должно быть больше нуля. Если в результате возникают проблемы, то данное правило можно отключить. Например, в `cypress-axe` можно исполнить команду `cy.configureAxe({rules: [{ id: 'tabindex', enabled: false }]})`, прежде чем выполнять аудит страницы.

Обсуждение

В нашем примере приложения продемонстрировано пользовательское диалоговое окно, чтобы стало ясно, в чем его недостатки. Используя готовые диалоговые окна и другие компоненты из таких библиотек, как Material UI, часто можно получить бесплатно многие другие возможности доступности. Кроме того, библиотеки часто создают всплывающие элементы вне пределов "корневого" раздела `<div>` приложения React. После этого они присваивают атрибуту `aria-hidden` всего "корневого" раздела `<div>` значение `true`, что практически скрывает все остальное приложение от средств чтения с экрана и другого ПО обеспечения доступности.

Отличный пример модального компонента, удовлетворяющего требованиям стандартов доступности, представляет React Modal разработки команды ReactJS (**https://oreil.ly/2nI5x**).

Исходный код данного рецепта можно загрузить на веб-сайте GitHub по адресу **https://oreil.ly/0GfgA**.

9.9. Создание считывателя экрана посредством Speech API

ЗАДАЧА

Проверить доступность приложения можно с помощью множества разных инструментов, но по-настоящему трудно получить представление о том, как воспринимает

ваше приложение пользователь с особыми потребностями. Поэтому наилучший подход к созданию доступного приложения — привлекать к процессу создания и тестирования кода людей, которым необходимы устройства для обеспечения доступности специальных возможностей.

Что касается остальных, то получить представление об использовании приложения посредством ПО для обеспечения доступности будет тем не менее полезным. Но с этим есть проблемы. Считыватели с выводом шрифта Брайля рассчитаны на умение пользователя читать этот шрифт. Программное обеспечение с голосовым выходом будет хорошей опцией, но большинство таких считывателей экрана довольно дорогие. Компьютеры Mac имеют встроенный считыватель экрана VoiceOver, содержащий множество возможностей, позволяющих перемещаться по экрану. Но компьютеры Mac есть не у всех.

Браузер Chrome снабжен хорошо работающим расширением ChromeVox, но оно поддерживается только этим браузером и, похоже, больше активно не разрабатывается.

Кроме всех этих моментов, считыватель экрана будет считывать все, что отображается на экране. Например, вы можете запустить считыватель экрана лишь для того, чтобы проверить на доступность какую-либо часть своего приложения, но он будет продолжать считывать экран и после того, как вы переключитесь обратно в среду разработки или откроете какой-либо справочный материал в другой вкладке браузера.

Учитывая все перечисленное, полезно попытаться самому испробовать звуковую версию своего приложения. По крайней мере, это даст вам определенное представление о том, насколько плохо большинство из нас справляются с задачей создания программного обеспечения, с которым могут работать люди с ограниченными возможностями.

Так как же нам можно испытать свое приложения со считывателем экрана?

РЕШЕНИЕ

Мы решим эту задачу, создав свой простой считыватель экрана. Очень, очень простой считыватель экрана. Это не будет считыватель профессионального качества, но он даст некоторое представление о том, как ваше приложение работает только с клавиатурой и голосовой обратной связью. Он также будет функционировать только с локальным приложением React, но не с другими страницами браузера или запущенными на компьютере приложениями. Этот считыватель называется TalkToMe[10].

Мы добавим небольшой фрагмент кода в приложение создания и редактирования задач, которое рассматриваем в качестве примера в этой главе. Мы не хотим включать код считывателя экрана в производственную версию приложения, поэтому начнем с добавления файла talkToMe.js в главную папку приложения. Содержимое этого файла приводится в листинге 9.50.

---

[10] Выражаем свою благодарность Терри Тиббсу (Terry Tibbs) за его помощь в создании этого инструмента.

Листинг 9.50. Содержимое файла talkToMe.js

```
function talkToMe() {
  if (
    process.env.NODE_ENV !== 'production' &&
    sessionStorage.getItem('talkToMe') === 'true'
  ) {
    ...
  }
}
```

Проверяя значения переменной `NODE_ENV`, можно ограничить исполнение кода нашей средой разработки. Мы также проверяем переменную сессионного хранилища `talkToMe`. Мы будем исполнять считыватель экрана только тогда, когда значение этой переменной равно `true`.

Нам нужен код для считывания подробной информации о текущем элементе с фокусом. События фокуса не передаются с нижнего уровня на верхний пузырьковым способом, поэтому мы не можем просто присоединить обработчик события `onFocus` к какому-либо элементу высшего уровня и начать отслеживать фокус.

Но мы в состоянии отслеживать события `focusin`. Мы можем прикрепить слушатель событий `focusin` к объекту `document`, и он будет вызываться при каждом переходе пользователя на новый компонент. Соответствующий код приведен в листинге 9.51.

Листинг 9.51. Добавление слушателя событий focusin

```
function talkToMe() {
  if (
    process.env.NODE_ENV !== 'production' &&
    sessionStorage.getItem('talkToMe') === 'true'
  ) {
    document.addEventListener('focusin', (evt) => {
      if (sessionStorage.getItem('talkToMe') === 'true') {
        ....
      }
    })
  }
}
```

Обратите внимание на то, что мы выполняем дополнительную проверку на наличие компонента `talkToMe`, просто на случай, если пользователь отключил его в процессе работы с приложением.

Нам нужно каким-либо образом описывать текущий элемент, имеющий фокус. Решить эту задачу нам поможет функция `getDescription()` (листинг 9.52), которая будет предоставлять общее описание текущего элемента на основе его названия, роли и т. п.

Листинг 9.52. Функция getDescription()

```
function getDescription(element) {
   const nodeName = element.nodeName.toUpperCase()
   const role = element.role
      ? element.role
      : nodeName === 'BUTTON'
      ? 'button'
      : nodeName === 'INPUT' || nodeName === 'TEXTAREA'
      ? 'text field ' + element.value
      : nodeName === 'SELECT'
      ? 'select field ' + element.value
      : element.getAttribute('role') || 'group'
   const title = element.title || element.textContent
   const extraInstructions =
      nodeName === 'INPUT' || nodeName === 'TEXTAREA'
         ? 'You are currently in a text field. To enter text, type.'
         : ''
   return role + '. ' + title + '. ' + extraInstructions
}
```

Теперь мы можем получить описание текущего элемента, имеющего фокус, как показано в листинге 9.53.

Листинг 9.53. Получение описания текущего элемента, имеющего фокус

```
function talkToMe() {
   if (
      process.env.NODE_ENV !== 'production' &&
      sessionStorage.getItem('talkToMe') === 'true'
   ) {
      document.addEventListener('focusin', (evt) => {
         if (sessionStorage.getItem('talkToMe') === 'true') {
            const description = getDescription(evt.target)
            ....
         }
      })
   }
}
```

Далее нам нужно преобразовать текст описания в речь. Для этого можно использовать API-интерфейс Web Speech, который в настоящее время поддерживают большинство браузеров. Синтезатор речи принимает в качестве параметра объект, называющийся *высказыванием* (utterance):

```
window.speechSynthesis.speak(
   new SpeechSynthesisUtterance(description)
)
```

Но прежде чем начинать читать фрагмент текста, нам нужно проверить, не читаем ли мы уже что-либо другое. Если читаем, то мы отменяем предыдущий процесс чтения и начинаем читать новый текст. Это позволит пользователю быстро переходить с одного компонента на другой, когда он решит, что услышал достаточно информации. Соответствующий код приведен в листинге 9.54.

Листинг 9.54. Отмена старого процесса чтения и начало нового

```
if (window.speechSynthesis.speaking) {
   window.speechSynthesis.cancel()
}
   window.speechSynthesis.speak(
      new SpeechSynthesisUtterance(description)
)
```

Теперь у нас есть все необходимые составляющие для создания конечной полной версии функции `talkToMe` (листинг 9.55).

Листинг 9.55. Конечная версия функции talkToMe

```
function talkToMe() {
   if (
      process.env.NODE_ENV !== 'production' &&
      sessionStorage.getItem('talkToMe') === 'true'
   ) {
      document.addEventListener('focusin', (evt) => {
         if (sessionStorage.getItem('talkToMe') === 'true') {
            const description = getDescription(evt.target)
            if (window.speechSynthesis.speaking) {
               window.speechSynthesis.cancel()
            }
            window.speechSynthesis.speak(
               new SpeechSynthesisUtterance(description)
            )
         }
      })
   }
}
```

Чтобы использовать функцию `talkToMe` в нашем приложении, мы вызываем ее в файле index.js в самом начале кода приложения, как показано в листинге 9.56.

Листинг 9.56. Вызов функции talkToMe в приложении

```
import React from 'react'
import ReactDOM from 'react-dom'
import './index.css'
```

```
import App from './App'
import reportWebVitals from './reportWebVitals'
import talkToMe from './talkToMe'
talkToMe()
ReactDOM.render(
   <React.StrictMode>
      <App />
   </React.StrictMode>,
   document.getElementById('root')
)
// If you want to start measuring performance in your app, pass a function
// to log results (for example: reportWebVitals(console.log))
// or send to an analytics endpoint. Learn more: https://bit.ly/CRA-vitals
(Если вы хотите выполнять замеры поддерживаемых метрик производительности в своем
приложении, передайте соответствующую функцию в функцию reportWebVitals(), чтобы
сохранить результаты в журнале (например, reportWebVitals(console.log)), или же
отправьте ее на узел аналитики(Google).
Дополнительную информацию см. по адресу: https://bit.ly/CRA-vitals)
reportWebVitals()
```

Теперь запустите наше приложение в браузере, откройте консоль разработчика, создайте новую переменную сессионного хранилища `TalkToMe` и присвойте ей значение `true`. В результате браузер должен голосом описывать элементы, между которыми вы перемещаетесь нажатиями клавиши <Tab>.

Обсуждение

Наш считыватель экрана `talkToMe` не более чем просто игрушка, но он поможет вам создавать краткие заголовки и другие метаданные в коде, подчеркивая, насколько важно размещать информацию в начале описаний. Чем быстрее пользователь примет решение, что он не заинтересован в данном элементе, тем раньше он сможет пойти дальше. Рассмотренный пример также позволит вам выяснить, какие части вашего приложения представляют трудности для навигации, и даст вам возможность испытать работу вашего приложения без использования экрана.

Исходный код данного рецепта можно загрузить на веб-сайте GitHub по адресу **https://oreil.ly/0GfgA**.

ГЛАВА 10

Производительность

Однажды преподаватель по информатике у одного из наших авторов начал урок с такой рекомендации: "Никогда, ни при каких обстоятельствах, ни за что в мире не оптимизируйте свой код. Но уж если вы оптимизируете свой код, то делайте это вот таким образом".

Как однажды сказал Дональд Кнут, преждевременная оптимизация — это корень всего зла. Лучше всего сначала добиться правильного функционирования кода, а затем работать над тем, чтобы сделать его поддающимся обслуживанию. И только тогда — и только при наличии проблемы — следует думать о том, как ускорить работу программы. Медленный код, который работает, всегда намного лучше быстрого кода, который не работает.

Сказав все это, заметим, что в некоторых обстоятельствах производительность может представлять значительную проблему. Если вашему приложению для загрузки требуется больше, чем несколько секунд, вы можете потерять некоторых пользователей, которые не станут ожидать так долго и уйдут со страницы, чтобы никогда больше на нее не возвратиться. На маломощных устройствах медленное приложение может превратиться в такое, с которым просто невозможно работать. В этой главе применяется, как мы его любим называть, эссенциалистский подход к производительности. Не следует часто выполнять настройку кода, но когда такая процедура выполняется, то только для кода, для которого она требуется. Мы рассмотрим различные инструменты и методы, которые позволят вам обнаруживать и оценивать узкие места в производительности, чтобы при необходимости исправить проблемы производительности, эти корректировки окажутся в правильном месте, и у вас будет какой-либо способ для измерения разницы в производительности, полученной вследствие этих исправлений.

Все исправления проблем производительности имеют свою цену. Если вы хотите ускорить исполнение кода клиента, это может стоить вам дополнительной памяти или серверного времени. И почти всегда придется увеличивать объем кода и его сложность.

Рецепты в этой главе следуют в том порядке, в котором мы рекомендуем подходить к решению проблем производительности. Мы начинаем с высокоуровневых измерений в браузере, и рассматриваем способы, посредством которых можно объективно идентифицировать узкие места производительности. Для обнаруженных узких мест мы покажем, как использовать встроенный в React компонент `Profiler` для локализации отдельных компонентов, являющихся причиной проблемы. Затем

мы рассмотрим низкоуровневые и более точные способы измерения производительности вплоть до величин меньше одной миллисекунды.

Только тогда, когда вы сможете точно измерять производительность, можно начинать думать о повышении скорости работы своего кода.

Затем мы покажем вам всего лишь несколько способов, которые могут улучшить производительность ваших приложений. Некоторые из них простые, например, разделение кода на пакеты меньшего размера или объединение асинхронных сетевых вызовов. Другие же более сложные, например предварительная отрисовка страниц на сервере.

В общем, эта глава намного больше об измерении производительности, нежели о настройке производительности. Поскольку вы никогда, ни при каких обстоятельствах, ни за что в мире не должны оптимизировать свой код, но когда оптимизируете, то нужно начинать с ее измерения.

10.1. Браузерные средства настройки производительности

ЗАДАЧА

Целесообразно повременить с настройкой производительности до тех пор, пока не будет видно, что в этой области есть проблема. Проблема имеется только в том случае, если пользователь видит, что приложение не работает должным образом. Но если откладывать решение проблемы до тех пор, пока пользователь заметит ее, то решение может быть слишком запоздалым. Поэтому желательно иметь определенный объективный критерий, показывающий, когда приложение нуждается в настройке, нечто, что реалистически измеряет производительность, а не просто определяет код, функционирование которого можно ускорить. Ускорить исполнение кода можно всегда, и разработчики зачастую тратят многие часы на настройку кода, не получив в результате никакого ощутимого эффекта с точки зрения восприятия пользователя.

Было бы полезным иметь в своем распоряжении какой-либо инструмент, способный выделять фрагменты кода, потенциально требующие оптимизации.

РЕШЕНИЕ

Наилучший способ проверить производительность — прибегнуть к помощи браузера. В конечном итоге единственное, что важно для нас, — это восприятие приложения пользователем. Поэтому мы рассмотрим разные браузерные инструменты, способные предоставить объектные измерения и обнаружить потенциальные узкие места в коде.

Первым из таких инструментов мы рассмотрим инструмент браузера Chrome, называющийся Lighthouse.

Компания Google также предоставляет встроенный модуль оценки производительности для браузера Firefox. Этот инструмент называется Google Lighthouse. Хотя данный инструмент хорошо работает, это просто клиентская часть службы Google Page Speed, и поэтому его можно использовать только на общедоступных веб-страницах. Но расширение Lighthouse браузера Chrome пригодно для любой страницы, которую браузер может прочитать.

Расширение Lighthouse — это отличное средство для проверки базовой пригодности вашего приложения. Кроме проверки производительности веб-страницы этот инструмент также проверяет ее доступность и соблюдение передовых практик в области веб-разработки. В частности, он проверяет оптимизацию страниц для роботов поисковых систем, а также соответствие веб-приложения требованиям стандартов, чтобы считаться прогрессивным веб-приложением (рис. 10.1).

Рис. 10.1. Метрики, проверяемые средством Lighthouse

Проверку посредством Lighthouse можно выполнять двумя способами: или из командной строки, или в браузере.

Для работы в командной строке, сначала необходимо установить инструмент `Lighthouse`:

```
$ npm install -g lighthouse
```

Затем можно выполнять проверку:

```
$ lighthouse http://localhost:3000
```

Версия командной строки средства Lighthouse — это просто автоматизированный сценарий для браузера Google Chrome. Ее преимущество состоит в том, что она создает HTML-отчет о проверке, что позволяет использовать ее на сервере непрерывной интеграции.

Со средством Lighthouse можно также работать интерактивно в браузере Google Chrome. Делать это лучше всего в окне инкогнито, чтобы снизить вероятность вмешательства в работу Lighthouse других расширений и хранилищ. Запустите браузер Chrome, откройте в нем свое приложение, откройте окно **Инструменты разработчика**, а в нем выберите вкладку **Lighthouse** (рис. 10.2).

На вкладке **Lighthouse** нажмите кнопку **Generate audit**. Средство Lighthouse начнет выполнять ряд проверок страницы, обновляя ее несколько раз в процессе работы. Аудит производительности концентрируется на шести метриках (рис. 10.3).

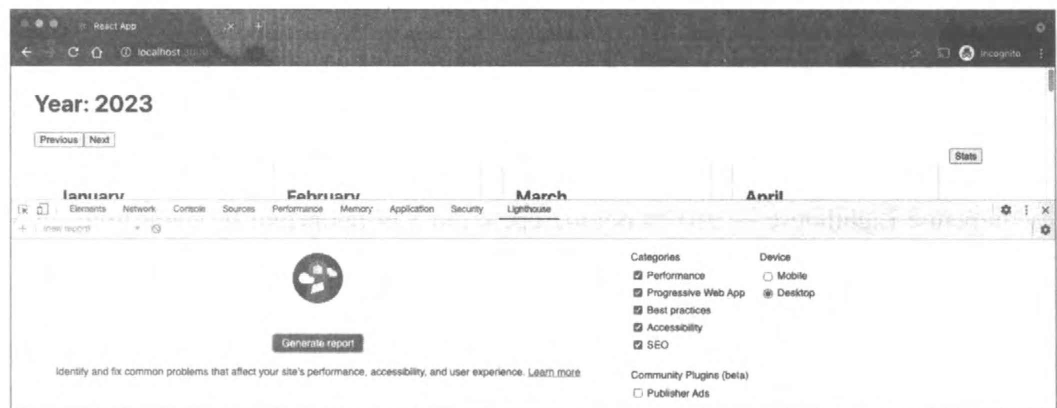


Рис. 10.2. Вкладка **Lighthouse** окна **Инструменты разработчика** браузера Chrome

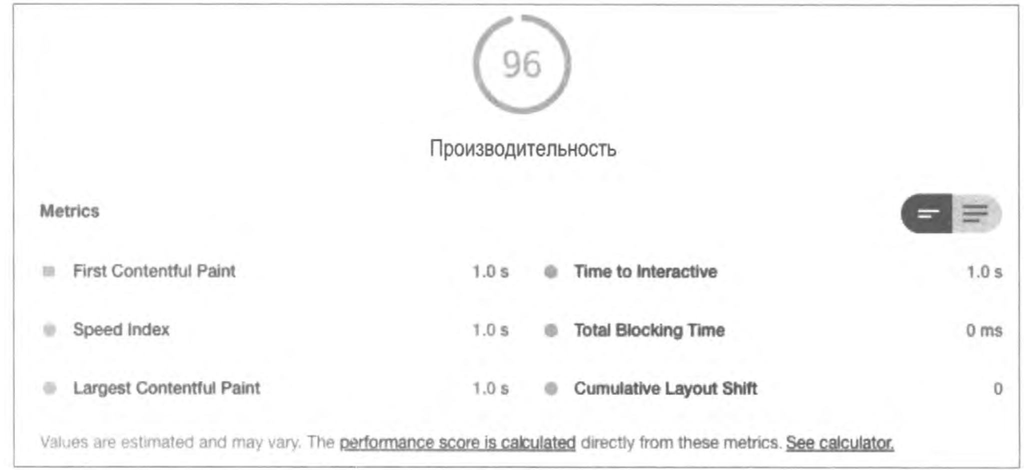


Рис. 10.3. Шесть важных метрик веб-приложения, измеряемые при аудите производительности Lighthouse

Эти метрики называются *основными показателями веб-производительности* (web vitals). Они позволяют отслеживать производительность приложений при их исполнении в рабочем режиме.

- **First Contentful Paint**, FCP (Первое отображение содержимого). Время, требуемое браузеру, чтобы начать отображать содержимое страницы после открытия ее пользователем. Метрика FCP сильно влияет на восприятие производительности пользователем. До начала отображения содержимого виден только пустой экран, и если это длится слишком долго, пользователь может закрыть эту страницу и перейти на какую-либо другую.

  Средство Lighthouse измеряет время, истекшее до FCP, а затем сравнивает полученное значение с соответствующими глобальными статистическими оценками производительности, сохраненными Google. Если данный показатель для вашего приложения находится в верхних 25% глобальных значений FCP, то он будет

зеленого цвета. В настоящее время ранжирование зеленым цветом означает, что первое содержимое отображается в течение двух секунд. Показатели в верхних 75% глобальных значений выделяются оранжевым цветом, такие страницы начинают отображаться в течение четырех секунд. Все другие значения этого показателя отображаются красным цветом.

- **Speed Index**, SI (Показатель быстродействия). Время, требуемое для визуальной стабилизации страницы. Тест выполняется путем записи видео загрузки страницы и проверки на наличие отличий между кадрами.

  Метрика SI сравнивается с соответствующими глобальными статистическими показателями. Значение SI до 4,3 секунды будет в верхних 25% глобальных показателей и отображается зеленым цветом, а до 5,8 секунды будет в верхних 75% и выделяется оранжевым цветом. Все другие значения этого показателя отображаются красным цветом.

- **Largest Contentful Paint**, LCP (Отображение крупнейшего элемента). Событие полной загрузки поля вывода браузера. При этом другое содержимое может загружаться в скрытом режиме, но LCP происходит, когда страница будет казаться видимой пользователю. Значения LCP до 2,5 секунды отображаются зеленым цветом, а до четырех секунд — оранжевым. Все остальные значения выделяются красным цветом. Отрисовка на стороне сервера может значительно улучшить рейтинг LCP.

- **Time to Interactive**, TTI (Время до интерактивности). Время, требуемое, чтобы со страницей можно было взаимодействовать посредством мыши и клавиатуры. В приложениях React это происходит после первой полной отрисовки, после присоединения React обработчиков событий. Для "зеленого" рейтинга этот показатель должен быть не большей 3,8 секунды, а для "оранжевого" — не больше 7,3 секунды. Все остальные значения получают "красный" рейтинг. Показатель TTI можно улучшить, отложив загрузку стороннего кода JavaScript или выполнив разделение кода[1].

- **Total Blocking Time**, TBT (Общее время блокировки). Сумма времени всех блокирующих задач, происходящих между событиями FCP и TTI. *Блокирующей задачей* считается все, что занимает свыше 50 мс. Это приблизительно время, требуемое для отображения кадра кинофильма, и все, что длится больше 50 мс, становится заметным. Если приложение имеет слишком много блокирующих задач, браузер начнет тормозить. По этой причине показатель TBT измеряется короткими периодами. Для "зеленого" рейтинга значение показателя TBT должно быть не больше 300 мс, для "оранжевого" — не больше 600 мс, а все свыше будет красным. При высоком значении показателя TBT пользователь будет испытывать ощущение, как будто бы браузер перегружен. Показатель TBT обычно можно улучшить, уменьшив объем исполняемого кода JavaScript или сократив количество сканирований модели DOM. Но наиболее эффективным методом будет, наверное, разделение кода.

---

[1] См. *раздел 10.5.*

♦ **Cumulative Layout Shift**, CLS (Совокупное смещение макета) Измерение смещения содержимого или визуального оформления страницы. Если во время загрузки страницы приложение вставляет дополнительное содержимое в уже загруженное содержимое, вызывая тем самым смещение части загруженного содержимого, то это будет отражаться на метрике CLS. Показатель CLS — это пропорциональная часть страницы, которая смещается в процессе загрузки.

В отчет Lighthouse не входит показатель *First Input Delay*, FID (Время ожидания до первого взаимодействия с содержимым). Этот показатель измеряет время между отправкой пользователем события странице (например, щелчок мышью) и получением этого события обработчиком JavaScript. Желаемое значение данного показателя не должно превышать 300 мс. Показать FID тесно связан с показателем TBT, поскольку блокирующие события обычно создаются обработчиками событий.

Кроме значений основных метрик страницы отчет Lighthouse также содержит рекомендации, как исправить обнаруженные им проблемы.

Использование Lighthouse будет хорошей начальной точкой проверки разрабатываемого приложения на наличие проблем производительности. Он не выполняет исчерпывающую проверку, но укажет на проблемы, которые вы сами могли бы и не заметить.

На результаты работы Lighthouse могут воздействовать многие факторы (пропускная способность сети, объем памяти, центральный процессор и т. п.), поэтому следует ожидать варьирующиеся результаты от одного прогона к другому. Онлайновые службы, такие как WebPageTest (**https://www.webpagetest.org**) или GTmetrix (**https://gtmetrix.com**), могут выполнять проверку приложения с разных местонахождений, что позволит получить более реалистичное представление о скорости работы вашего приложения, чем предоставляемое Lighthouse, выполняющим проверку по адресу **http://localhost:3000**.

Инструмент Lighthouse хорошо справляется с обнаружением проблем производительности, но выявить причины этих проблем способен в меньшей степени. А эти причины могут быть разными: чрезмерный объем кода страницы или его слишком медленное исполнение, медленно отвечающий сервер, или даже проблема с аппаратными ресурсами, такая как недостаточный объем памяти или большой объем кеша.

Чтобы выяснить причину узкого места, на следующем шаге можно прибегнуть к средствам производительности самого браузера.

В браузерах Firefox и Chrome консоль производительности можно открыть, открыв исследуемую страницу в окне инкогнито, а затем выбрав вкладку **Performance** в панели инструментов разработчика (рис. 10.4).

Вкладку **Performance** можно сравнить с системой управления двигателем для браузера. На ней можно отслеживать использование памяти, блокирующее центральный процессор задачи, число элементов в модели DOM и т. п. Для сбора статистических данных по производительности страницы нужно нажать кнопку **Record** на панели инструментов, а затем взаимодействовать со страницей в течение нескольких секунд, после чего остановить запись. Система тестирования производительности

выполнит оценку всех указанных вами составляющих. В примере на рис. 10.5 показано возникновение блокирующей операции (см. описание показателя ТВТ ранее в этом разделе) при нажатии кнопки пользователем, и браузер был заблокирован в течение 60,92 мс, пока обработчик события не возвратил управление.

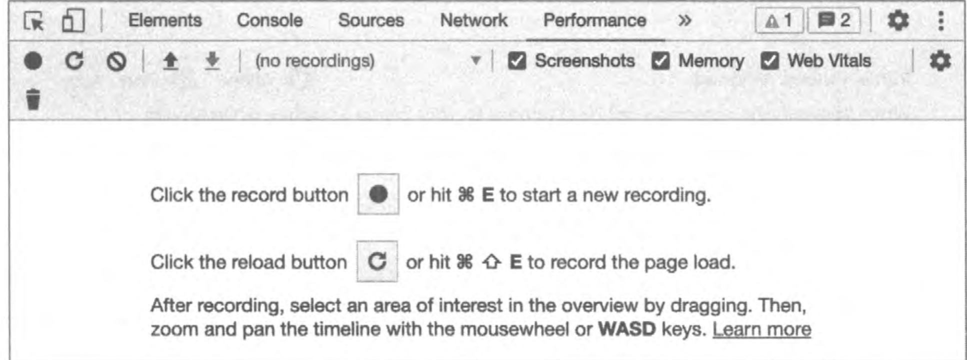


Рис. 10.4. Вкладка **Performance** панели **Инструменты разработчика** браузера

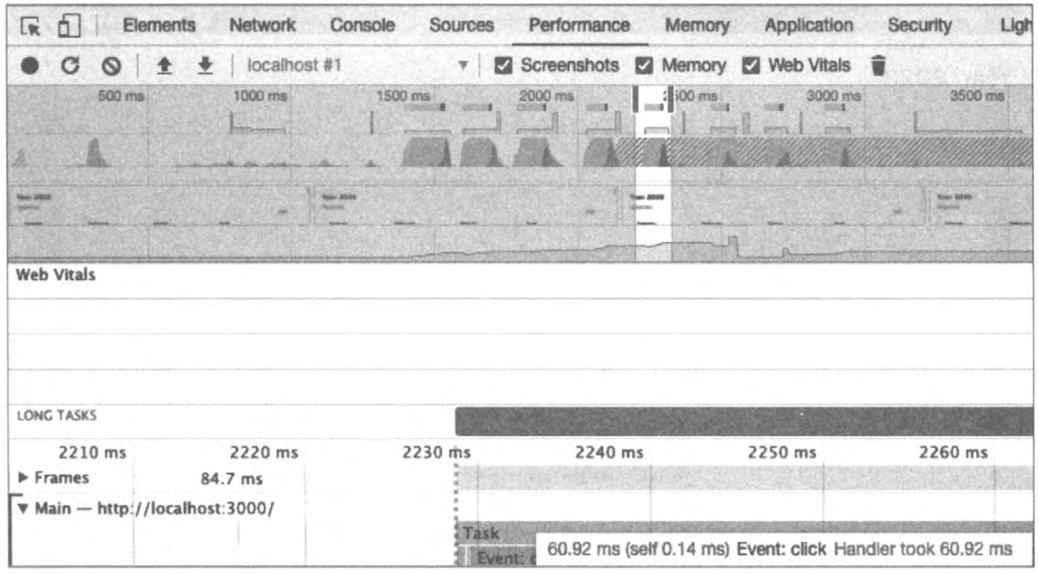


Рис. 10.5. Увеличение масштаба, чтобы исследовать долго исполняющуюся задачу

Вкладка **Performance** предоставляет все статистические данные, которые вам когда-либо могут потребоваться при настройке производительности. Более того, она, вероятно, содержит намного больше подробной информации, чем вам потребуется. По этой причине целесообразнее установить набор инструментов React Developer Tools для Chrome (**https://oreil.ly/vvCLp**) или Firefox (**https://oreil.ly/mw1yn**).

Может случиться, что после установки эти инструменты по умолчанию не работают в режиме инкогнито. Будет полезным разрешить этот режим, как показано на рис. 10.6 для Chrome и на рис. 10.7 для Firefox.

Allow in incognito

Warning: Google Chrome cannot prevent extensions from recording your browsing history. To disable this extension in incognito mode, unselect this option.

Рис. 10.6. Разрешение режима инкогнито для набора Developer Tools в Chrome

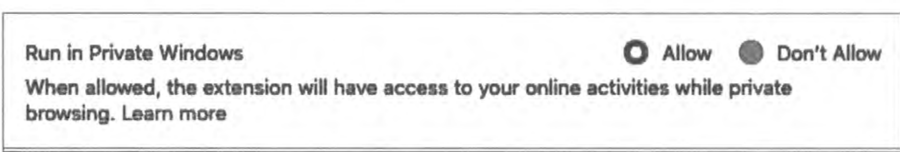


Рис. 10.7. Разрешение режима приватности для набора Developer Tools в Chrome

Подобно инструментам оценки производительности браузера в наборе React Developer Tools для записи сеанса производительности нужно нажать кнопку **Record** в левом верхнем углу панели инструментов (рис. 10.8).

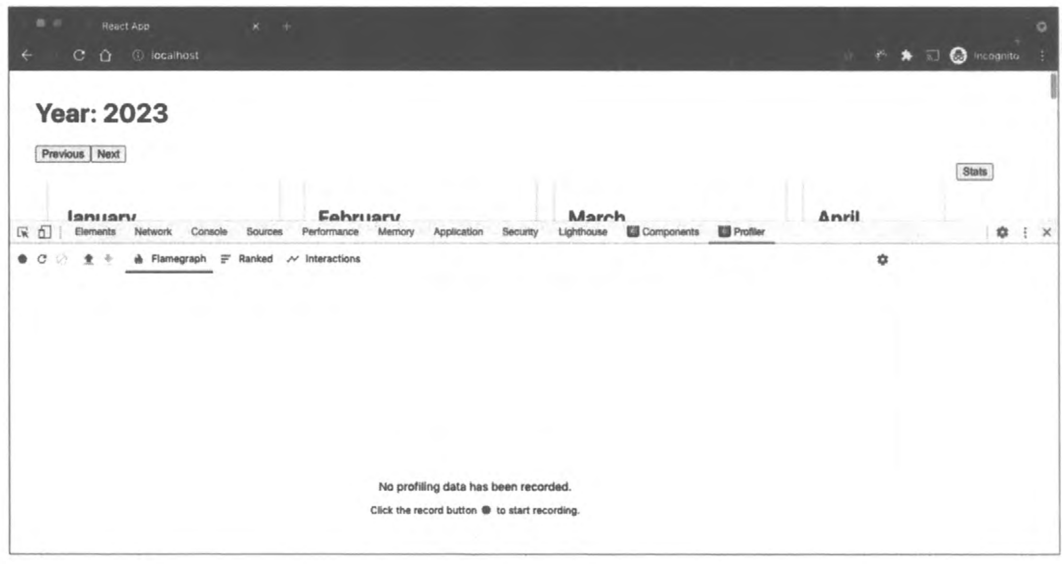


Рис. 10.8. Вкладка **React Profiler** в наборе Developer Tools для Chrome

После записи сеанса отображаются статистические данные по производительности, а также связанные с React компоненты, которые отрисовали веб-страницу. Если отображение какого-либо компонента заняло длительное время, то выяснить, какой это компонент, можно, наведя в графе результатов оценки производительности курсор мыши на его данные, в результате чего соответствующий компонент будет выделен (рис. 10.9).

Во многих случаях набор инструментов React Developer Tools оказывается наилучшим интерактивным средством для определения причины проблемы производительности. Но, как уже упоминалось ранее, рассматривать возможность настройки

производительности следует только при обнаружении узкого места в производительности пользователем или каким-либо высокоуровневым инструментом наподобие Lighthouse.

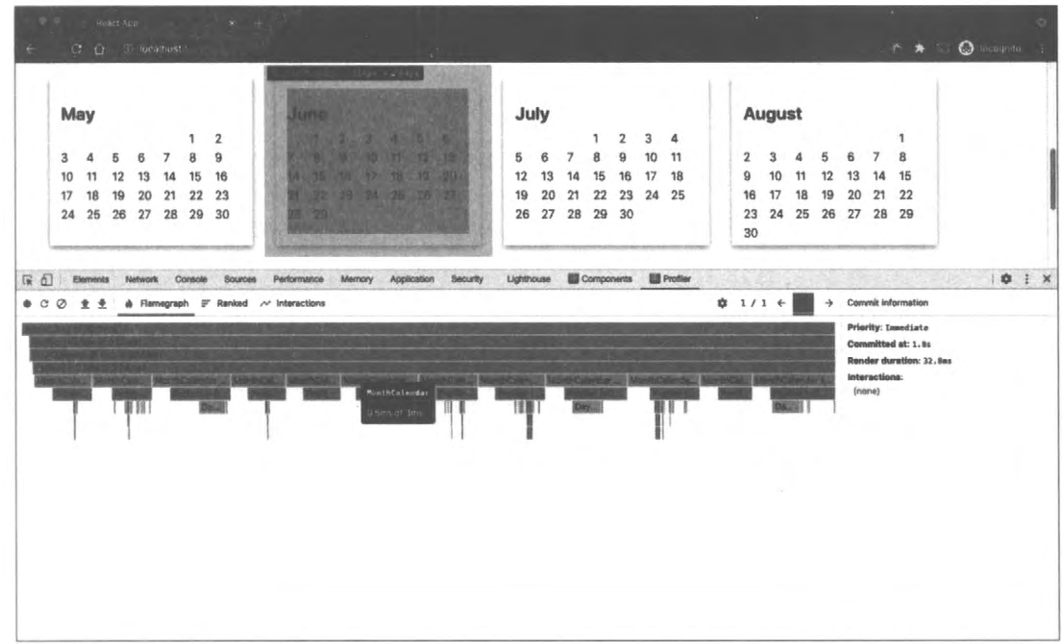


Рис. 10.9. Наведение курсора мыши на данные компонента в графике результатов оценки производительности выделяет этот компонент

Обсуждение

В случае применения эссенциалистского подхода к настройке производительности всегда следует начинать с браузера, или используя само приложение или какой-либо из встроенных инструментов и расширений, рассмотренных ранее в этом разделе.

10.2. Слежение за отрисовкой посредством *Profiler*

ЗАДАЧА

Браузерные инструменты предоставляют подробную информацию о производительности, и их нужно всегда задействовать на первом этапе при определении причин проблем производительности.

После определения проблемы полезно получить более подробные статистические данные для небольшой части приложения. Единственный способ повысить производительность — сбор фактических данных производительности до и после моди-

фикации кода. Это может быть трудно реализовать при помощи браузерных расширений, поскольку они "заваливают" данными обо всех аспектах работы приложения.

Как же нам получить статистические данные только для той части приложения, настройку которой мы выполняем?

РЕШЕНИЕ

Для решения сформулированной задачи можно использовать компонент React `Profiler`. В этот компонент можно заключить любую часть приложения, для которой выполняется настройка. После этого он будет записывать статистические данные при каждой его отрисовке. В табл. 10.1 приведен список и краткое описание собираемых таким образом важных данных.

Таблица 10.1. *Данные, записываемые компонентом Profiler*

Показатель	**Назначение**
Phase (Этап)	Что вызвало отрисовку· монтирование или обновление
Actual duration (Фактическая длительность)	Сколько времени заняла бы полная отрисовка, если бы не применялось внутреннее кеширование
Base duration (Базовая длительность)	Сколько времени заняла отрисовка с применением кеширования
Start time (Время начала)	Число миллисекунд, истекших после загрузки страницы
Commit time (Время фиксации)	Когда результаты отрисовки прибывают в модель DOM браузера
Interactions (Взаимодействия)	Обработчики событий, отслеживаемые в настоящее время

Чтобы разобраться с работой компонента `Profiler`, начнем с рассмотрения примера приложения, окно которого приведено на рис. 10.10.

В листинге 10.1 приведен код компонента `App` этого приложения.

Листинг 10.1. Код компонента App примера приложения Calendar

```
import { useState } from 'react'
import { unstable_trace as trace } from 'scheduler/tracing'
import './App.css'
function App({ onRender }) {
   const [year, setYear] = useState(2023)
   return (
      <div className="App">
         <h1>Year: {year}</h1>
         <button onClick={() => setYear((y) => y - 1)}>Previous</button>
         <button onClick={() => setYear((y) => y + 1)}>Next</button>
         <br />
```

```
            <YearCalendar year={year} onRender={onRender} />
        </div>
    )
}
export default App
```

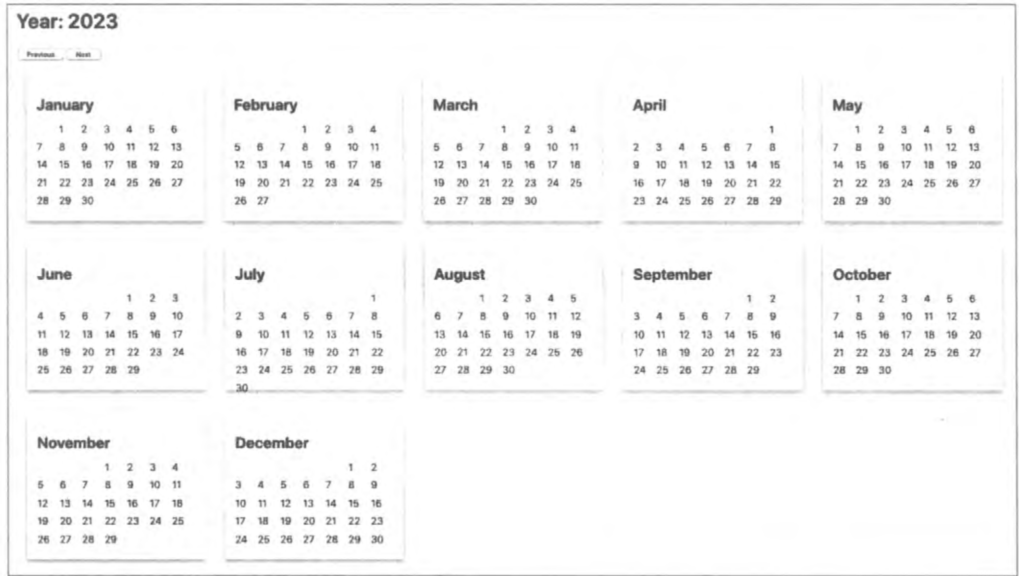


Рис. 10.10. Окно примера приложения Calendar

Наше приложение отображает две кнопки: одна для прямого перемещения по годам, а вторая — для обратного.

Сначала заключим эти две кнопки и компонент календаря в оболочку компонента `Profiler`, как показано в листинге 10.2.

Листинг 10.2. Заключение кнопок в оболочку компонента `Profiler`

```
import { useState, Profiler } from 'react'
import { unstable_trace as trace } from 'scheduler/tracing'
import './App.css'

function App({ onRender }) {
    const [year, setYear] = useState(2023)
    return (
        <div className="App">
            <h1>Year: {year}</h1>
            <Profiler id="app" onRender={() => {}}>
                <button onClick={() => setYear((y) => y - 1)}>
                    Previous
                </button>
```

```
            <button onClick={() => setYear((y) => y + 1)}>Next</button>
            <br />
            <YearCalendar year={year} onRender={onRender} />
         </Profiler>
      </div>
   )
}
export default App
```

Компонент `Profiler` принимает в качестве параметров переменную `id` и функцию обратного вызова `onRender`. При каждой отрисовке компонента `Profiler` он отправляет статистические данные функции `onRender`. Поэтому добавим еще подробностей о функции `onRender` (листинг 10.3).

Листинг 10.3. Дополнительные подробности о функции onRender

```
import { useState, Profiler } from 'react'
import { unstable_trace as trace } from 'scheduler/tracing'
import './App.css'
let renders = []
   let tracker = (
   id,
   phase,
   actualDuration,
   baseDuration,
   startTime,
   commitTime,
   interactions
) => {
   renders.push({
      id,
      phase,
      actualDuration,
      baseDuration,
      startTime,
      commitTime,
      interactions: JSON.stringify(Array.from(interactions)),
   })
}
function App({ onRender }) {
   const [year, setYear] = useState(2023)
   return (
      <div>
         ....
         <Profiler id="app" onRender={tracker}>
            ....
         </Profiler>
```

```
        <button onClick={() => console.table(renders)}>Stats</button>
      </div>
    )
}
```

Функция `tracker` записывает все результаты, получаемые от компонента `Profiler`, в массив `renders`. Мы также добавили в интерфейс кнопку **Stats**, нажатие которой будет отображать в консоли таблицу параметров отрисовки.

Перезагрузим страницу и нажмем несколько раз кнопки **Previous** и **Next**, а затем кнопку **Stats**. В результате в консоли отобразятся статистические данные профиля (рис. 10.11).

Year: 2022

Previous Next

Stats

Elements Network Console Sources Performance Memory Application Security Lighthouse Firebase Redux Components Profiler axe DevTools

top Filter Default levels No Issues 1 hidden

[HMR] Waiting for update signal from WDS... log.js:24 App.js:48

(index)	id	phase	actualDuration	baseDuration	startTime	commitTime	interactions
0	"app"	"mount"	301.71999998856336	201.20500022312626	1494.2300000111572	1815.614999909525	"[]"
1	"app"	"update"	124.11000000042934	85.14000018476509	5796.18000000004824	5937.00000014249	"[]"
2	"app"	"update"	132.39500019699335	89.19999990030192	5990.280000027269	6126.830000022892	"[]"
3	"app"	"update"	119.42499977085745	89.01999972295016	6177.07500002305	6299.40000018068	"[]"
4	"app"	"update"	64.12999986787327	47.32000001870282	6329.24500000081	6395.595000000525	"[]"
5	"app"	"update"	64.50000055863522	46.73000043840147 6	6474.205000005095	6541.605000005082	"[]"
6	"app"	"update"	68.60000005690381	50.93500032671727	6842.350000020815	6913.050000020405	"[]"
7	"app"	"update"	54.20500010950401	40.70500057423487	8459.240000025602	8515.16500001800 7	"[]"
8	"app"	"update"	63.92000001505948 6	38.97000002325512 5	8617.760000022827	8683.40000000098 7	"[]"
9	"app"	"update"	55.28499963111244	36.57000014209188 5	8793.640000010058	8850.790000025081	"[]"
10	"app"	"update"	46.13000017707236	35.93000018736347 6	8970.085000000112	9017.930000001797	"[]"
11	"app"	"update"	67.56499991752207	54.46499970275909	9138.72500002617	9210.32000001287	"[]"
12	"app"	"update"	37.18000007211231	29.09000017090151	9313.135000003967	9351.755000010598	"[]"
13	"app"	"update"	46.92500029341318	31.57500037923455 2	9474.215000023833	9522.635000001173	"[]"

Рис. 10.11. Статистика отрисовок, выводимая в консоль JavaScript

Эти данные выводятся в табличном формате, что немного облегчает работу с ними, а также позволяет упорядочивать их по любому столбцу. Можно скопировать всю информацию и вставить ее в электронную таблицу для дополнительного анализа.

Обратите внимание на то, что столбец `interactions` всегда содержит пустой массив. Это объясняется тем, что в настоящее время мы не отслеживаем никаких обработчиков событий или других фрагментов кода. Чтобы увидеть, какие обработчики событий исполняются в процессе отрисовки в настоящее время, можно импортировать функцию отслеживания и облачить в нее любой фрагмент кода, который нужно отслеживать. Например, в листинге 10.4 показано, как можно начать отслеживать событие нажатия пользователем кнопки **Previous**.

Листинг 10.4. Код для отслеживания события нажатия кнопки Previous

```
import { unstable_trace as trace } from 'scheduler/tracing'
...
<button
  onClick={() => {
    trace('previous button click', performance.now(), () => {
```

```
        setYear((y) => y - 1)
      })
    }}
/>
```

Функция `trace` принимает в качестве параметров метку, отметку времени и функцию обратного вызова, содержащую отслеживаемый ею код. Параметр отметки времени мог бы быть датой, но часто будет лучше задать значение в миллисекундах, возвращаемое функцией `performance.now()`.

Перезагружаем страницу, нажимаем несколько раз кнопку **Next**, а затем кнопку **Previous**. В таблице результатов должны начать отображаться взаимодействия (interactions) в виде строк JSON (рис. 10.12).

top ▾ Filter Default levels ▾ 1 Issue: 1

1.html:3

(in…	id	phase	actualDur…	baseDuration	startTime	commitTime	iteractions
0	1	"mount"	447.54999…	234.53534542685	1333.65466546…	1811.765745675473	"[{\"__count\":1,\"id\":0…
1	1	"update"	153.98661…	87.767252662362	2433.65464562…	2478.7657657453	"[{\"__count\":1,\"id\":0…
2	1	"update"	115.54968…	71.543256572268	2565.65464642…	2579.7657564743	"[{\"__count\":1,\"id\":0…
3	1	"update"	134.98661…	120.65463563466	2954.66546642…	3014.765765453	"[{\"__count\":1,\"id\":0…
4	1	"update"	79.654634…	54.254252345685	3100.56476632…	3145.7654746743	"[{\"__count\":1,\"id\":0…
5	1	"update"	131.86876…	23.534636564656	3400.63654645…	3478.765745743	"[{\"__count\":1,\"id\":0…
6	1	"update"	66.999866…	43.68554325235	3601.87687682…	3654.76576547543	"[{\"__count\":1,\"id\":0…
7	1	"update"	105.61646…	100.535425685	3712.78782349…	3717.765765473	"[{\"__count\":1,\"id\":0…
8	1	"update"	61.543258…	42.653523453245	3922.87687872…	4014.745677453	"[{\"__count\":1,\"id\":0…
9	1	"update"	66.549998…	60.532523543268	4398.87876867…	4432.6546453653	"[{\"__count\":1,\"id\":0…
10	1	"update"	67.667854…	51.232543526585	5108.87878768…	5210.3654654334	"[{\"__count\":1,\"id\":0…

Рис. 10.12. Отображение отслеживаемых взаимодействий в таблице результатов

Выходные данные преобразуются в строки по той причине, что функция `trace` сохраняет взаимодействия в виде множеств JavaScript, которые часто не отображаются правильно в консоли. Хотя в таблице данные взаимодействий выглядят обрезанными, в действительности это не так, и при копировании можно получить полные данные. В листинге 10.5 приведен пример данных, возвращенных одним отслеженным взаимодействием.

Листинг 10.5. Данные одного отслеженного взаимодействия

```
[
  {
    "__count":1,
    "id":1,
    "name":"previous button click",
    "timestamp":4447.909999988042
  }
]
```

Обсуждение

Компонент `Profiler` был добавлен в React в версии 16.4.3. Функция `trace` все еще экспериментальная, но крайне мощная. Хотя в данном примере мы использовали ее лишь для простого обработчика события, она также может предоставлять временные данные для более крупных фрагментов кода, например для сетевых запросов. Контейнерные компоненты React в процессе отрисовки часто имеют большое число сетевых запросов "на лету", и функция `trace` предоставляет возможность увидеть, что происходило во время особенно медленной отрисовки. Она также может дать некоторое представление о количестве отрисовок, полученных из целой цепи разных сетевых процессов.

Исходный код данного рецепта можно загрузить на веб-сайте GitHub по адресу **https://oreil.ly/0GfgA**.

10.3. Создание модульных тестов с *Profiler*

ЗАДАЧА

Компонент React `Profiler` — это мощный инструмент. Он обеспечивает разработчику доступ к той же самой информации о ходе выполнения программы, которую предоставляет набор инструментов React Developer Tools. Преимущество `Profiler` состоит в том, что разработчик может сконцентрироваться на определенном коде, который он пытается оптимизировать.

Но данный компонент все равно полагается на ручные взаимодействия разработчика с веб-страницей. Производительность нужно измерять до и после модификации кода. Но как можно быть уверенным в том, что показания до и после модификации отображают те же самые характеристики? Если мы осуществляем тестирование вручную, как можно гарантировать, что каждый раз мы будем выполнять одинаковый набор действий?

РЕШЕНИЕ

В этом рецепте мы рассмотрим, как создавать модульные тесты, которые вызывают код с компонентом `Profiler`. Автоматизированные тесты позволят нам выполнять повторяемые проверки производительности, чтобы выяснить, насколько эффективными в плане производительности являются вносимые нами правки в код.

В модульном тесте мы можем отрисовать компонент React вне браузера благодаря предоставляемой библиотекой Testing Library "безголовой" реализации модели DOM.

Чтобы разобраться с использованием компонента `Profiler`, мы опять обратимся к примеру приложения календаря (рис. 10.13).

Мы можем добавить компонент `Profiler` в основной код компонента `App` нашего приложения, а затем позволить любому другому коду передавать функцию `onRender`,

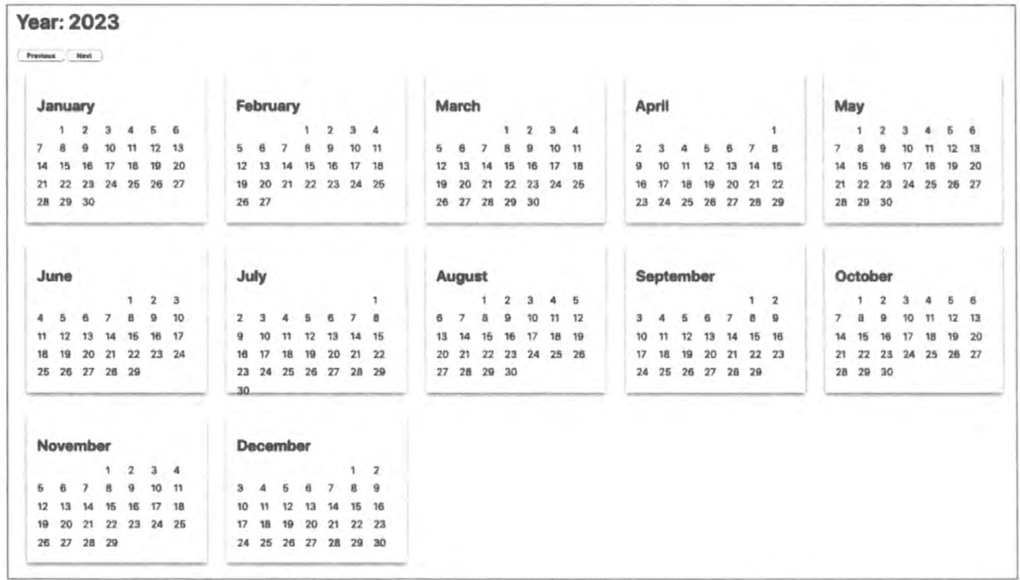


Рис. 10.13. Окно примера приложения Calendar

посредством которой можно отслеживать производительность отрисовки. Соответствующий код приведен в листинге 10.6.

Листинг 10.6. Добавление компонента `Profiler` в код компонента `App`

```
import { useState, Profiler } from 'react'
import YearCalendar from './YearCalendar'
import { unstable_trace as trace } from 'scheduler/tracing'
import './App.css'
function App({ onRender }) {
   const [year, setYear] = useState(2023)
   return (
      <div className="App">
         <h1>Year: {year}</h1>
         <Profiler id="app" onRender={onRender || (() => {})}>
            <button
               onClick={() => {
                  trace('previous button click', performance.now(), () => {
                     setYear((y) => y - 1)
                  })
               }}
            >
               Previous
            </button>
            <button onClick={() => setYear((y) => y + 1)}>Next</button>
            <br />
            <YearCalendar year={year} onRender={onRender} />
```

```
        </Profiler>
      </div>
    )
}
export default App
```

Мы также можем передавать функцию `onRender` вниз по иерархии компонентам-потомкам, чтобы отслеживать их производительность отрисовки. В коде в листинге 10.6 мы передаем функцию `onRender` компоненту `YearCalender`, который затем может или задействовать ее в своем компоненте `Profiler`, или передать ее далее вниз по дереву компонентов.

Можно избежать необходимости передавать функцию `onRender` компонентам-потомкам, создав компонент-поставщик, который будет инжектировать эту функцию в текущий контекст. В данном случае мы этого не делаем, чтобы не усложнять код. Но в других разделах в книге есть несколько примеров использования поставщиков. Например, компонент-поставщик `SecurityProvider` в *разделе 7.1*.

Компоненту `Profiler` необходимо передать свойство `id` и свойство `onRender`. При нормальном исполнении приложения компоненту `App` свойство `onRender` не передается, поэтому нужно предоставить функцию по умолчанию:

```
<Profiler id='app' onRender={onRender || (() => {})}>
```

Компонент `Profiler` сравнительно легковесный и обычно не замедляет производительность приложения. Если забыть удалить данный компонент из кода, то это не будет иметь большого значения. Компонент `Profiler` используется только в режиме разработки. Он удаляется из кода при создании рабочей версии приложения.

Теперь можно приступать к созданию модульного теста (листинг 10.7).

Листинг 10.7. Код модульного теста

```
import { render, screen } from '@testing-library/react'
import user from '@testing-library/user-event'
import App from './App'
let renders = []
let tracker = (
  id,
  phase,
  actualDuration,
  baseDuration,
  startTime,
  commitTime,
  interactions
) => {
  renders.push({
  id,
```

```
    phase,
    actualDuration,
    baseDuration,
    startTime,
    commitTime,
    interactions: JSON.stringify(Array.from(interactions)),
    })
}
let startTime = 0
describe('App', () => {
    beforeEach(() => {
        renders = []
        startTime = performance.now()
    })
    afterEach(() => {
        console.log('Time taken: ', performance.now() - startTime)
        console.table(renders)
    })
    it('should move between years', async () => {
        render(<App onRender={tracker} />)
        user.click(screen.getByRole('button', { name: /previous/i }))
        user.click(screen.getByRole('button', { name: /previous/i }))
        user.click(screen.getByRole('button', { name: /previous/i }))
        user.click(screen.getByRole('button', { name: /next/i }))
        user.click(screen.getByRole('button', { name: /next/i }))
        user.click(screen.getByRole('button', { name: /next/i }))
    }, 30000)
})
```

Тесты длительностью свыше пяти секунд, скорее всего, будут превышать предел тайм-аута Jest. Чтобы избежать этого, проще всего добавить параметр тайм-аута в вызов функции `it` и присвоить ему значение 30 000 мс (как это делаем мы в приведенном коде). Это значение нужно будет корректировать в соответствии со сложностью теста.

При исполнении этого теста в консоли отображается громадный объем данных (рис. 10.14).

Данный тест отличается своей повторяемостью, т. е. при каждом прогоне он будет выполнять одинаковые действия. Мы обнаружили, что модульные тесты, как правило, более единообразны, чем код, исполняемый в браузере. При многократном прогоне нашего теста время исполнения каждого прогона было 21 000±20 мс. Это разброс величиной менее чем 1%. И каждый прогон создал точно 2653 оценки профиля приложения.

Вряд ли бы мы получили повторяемые результаты при ручном тестировании в браузере.

В рассмотренном примере мы просто отображаем захваченные данные. При настоящем тестировании производительности лучше обработать результаты каким-

либо образом, чтобы, например, узнать среднее время обработки определенного компонента. Затем, когда вы начнете выполнять настройку этого компонента, можно быть более уверенным в том, что любые улучшения производительности получены вследствие фактических модификаций кода, а не в результате вариаций в работе браузера.

```
/Users/davidg/.nvm/versions/node/v12.20.0/bin/node --require "/Applications/IntelliJ IDEA.app/Contents/plugins/JavaScriptLanguage/helpers/jest-intellij/lib/jest-intellij-stdin-fix.js" /Users/davidg/Desktop/code/chapter10/calendar/app/node_modules/react-scripts/bin/react-scripts.js test --colors --reporters "/Applications/IntelliJ IDEA.app/Contents/plugins/JavaScriptLanguage/helpers/jest-intellij/lib/jest-intellij-reporter.js" --verbose "--testNamePattern=^App " --runTestsByPath /Users/davidg/Desktop/code/chapter10/calendar/app/src/App.test.js
  console.log
    Time taken:  2055.683333

      at Object.<anonymous> (src/App.test.js:34:17)

  console.log
```

(index)	id	phase	actualDuration	baseDuration	startTime	commitTime	interactions
0	'Day-2023-1-0'	'mount'	0.013593000000128086	0.011005400000068321	2029.129893	2244.32986	'[]'
1	'Day-2023-1-1'	'mount'	0.3774439999999600	0.11832000000002469	2030.638281	2244.32986	'[]'
2	'Day-2023-1-2'	'mount'	0.22844300000018757	0.056252999999996369	2031.472044	2244.32986	'[]'
3	'Day-2023-1-3'	'mount'	0.2320830000001824	0.052635000000000931	2032.05237	2244.32986	'[]'
4	'Day-2023-1-4'	'mount'	0.21129899999982626	0.051211999999985038	2032.640459	2244.32986	'[]'
5	'Day-2023-1-5'	'mount'	0.2061769999995704	0.052945999999979255	2033.21197	2244.32986	'[]'
6	'Day-2023-1-6'	'mount'	0.19538100000022496	0.050961000000014383	2033.700082	2244.32986	'[]'
7	'Day-2023-1-7'	'mount'	0.1915750000002645	0.051294000000000004	2034.308503	2244.32986	'[]'
8	'Day-2023-1-8'	'mount'	0.19614999999990872	0.051490000000005799	2034.842364	2244.32986	'[]'
9	'Day-2023-1-9'	'mount'	0.20954299999993964	0.050138999999959196	2035.400933	2244.32986	'[]'
10	'Day-2023-1-10'	'mount'	0.20773099999986703	0.052215999999981663	2035.982062	2244.32986	'[]'
11	'Day-2023-1-11'	'mount'	0.18697899999982747	0.054123999999977424	2036.5211	2244.32986	'[]'
12	'Day-2023-1-12'	'mount'	0.1721789999999146	0.050563000000001374	2036.99502	2244.32986	'[]'
13	'Day-2023-1-13'	'mount'	0.17159000000046946	0.050443000000212996	2037.48513	2244.32986	'[]'
14	'Day-2023-1-14'	'mount'	0.17261599999960709	0.049483999999969306	2037.958206	2244.32986	'[]'
15	'Day-2023-1-15'	'mount'	0.16483199999990793	0.048897999999990468	2038.390604	2244.32986	'[]'
16	'Day-2023-1-16'	'mount'	0.16570400000025795	0.049071000000016653	2038.819527	2244.32986	'[]'
17	'Day-2023-1-17'	'mount'	0.17260699999997087	0.051062000000000172	2039.245773	2244.32986	'[]'
18	'Day-2023-1-18'	'mount'	0.16747099999975035	0.050760999999973849	2039.670767	2244.32986	'[]'
19	'Day-2023-1-19'	'mount'	0.16577000000004572	0.051165000000000273	2040.18428	2244.32986	'[]'
20	'Day-2023-1-20'	'mount'	0.17082100000017454	0.055233000000004383	2040.554798	2244.32986	'[]'
21	'Day-2023-1-21'	'mount'	0.19743199999993521	0.055053999999992733	2041.028808	2244.32986	'[]'
22	'Day-2023-1-22'	'mount'	0.19727400000010675	0.051503000000130854	2041.529841	2244.32986	'[]'
23	'Day-2023-1-23'	'mount'	0.19566099999995762	0.051537000000005383	2042.038860	2244.32986	'[]'
24	'Day-2023-1-24'	'mount'	0.2023369999998704	0.050741000000016356	2042.681066	2244.32986	'[]'
25	'Day-2023-1-25'	'mount'	0.21615500000020897	0.049469000000044618	2043.301176	2244.32986	'[]'
26	'Day-2023-1-26'	'mount'	0.3055710000000996	0.06234099999999832	2043.853785	2244.32986	'[]'
27	'Day-2023-1-27'	'mount'	0.3479239999996035	0.057587999999996856	2044.632651	2244.32986	'[]'
28	'Day-2023-1-28'	'mount'	0.20145999999976993	0.06171799999999828	2045.34598	2244.32986	'[]'
29	'Day-2023-1-29'	'mount'	0.18842099999983475	0.050766000000005783	2045.080476	2244.32986	'[]'
30	'Day-2023-1-30'	'mount'	0.14869000000021515	0.046111999999988235	2046.31345	2244.32986	'[]'
31	'MonthCalendar'	'mount'	16.070473000001766	4.827159000000165	2020.313042	2244.32986	'[]'
32	'Day-2023-2-0'	'mount'	0.011032000000341213	0.008736000000027634	2049.09288	2244.32986	'[]'
33	'Day-2023-2-1'	'mount'	0.16821600000002945	0.044879999999992999	2049.904682	2244.32986	'[]'
34	'Day-2023-2-2'	'mount'	0.2862609999997403	0.11793200000010984	2050.365416	2244.32986	'[]'
35	'Day-2023-2-3'	'mount'	0.20043399999985922	0.055362999999350957	2050.978203	2244.32986	'[]'
36	'Day-2023-2-4'	'mount'	1.2877540000004774	0.006782000000045934	2051.637919	2244.32986	'[]'
37	'Day-2023-2-5'	'mount'	0.2097170000006372	0.052654999999993882	2053.258934	2244.32986	'[]'
38	'Day-2023-2-6'	'mount'	0.18125000000075792	0.050900000000006179	2053.759186	2244.32986	'[]'
39	'Day-2023-2-7'	'mount'	0.15399500000027328	0.045452900000029581	2054.188094	2244.32986	'[]'
40	'Day-2023-2-8'	'mount'	0.15200100000043414	0.047826000000044113	2054.587404	2244.32986	'[]'
41	'Day-2023-2-9'	'mount'	0.14911599999959435	0.044076999999980492	2054.088101	2244.32986	'[]'
42	'Day-2023-2-10'	'mount'	0.14841899999985253	0.045712999999997791	2055.379818	2244.32986	'[]'
43	'Day-2023-2-11'	'mount'	0.14682100000002014	0.04451999999999204	2055.773292	2244.32986	'[]'
44	'Day-2023-2-12'	'mount'	0.156481999999869	0.045776000000016328	2056.167921	2244.32986	'[]'
45	'Day-2023-2-13'	'mount'	0.14775799999915762	0.043572999999469175	2056.564458	2244.32986	'[]'
46	'Day-2023-2-14'	'mount'	0.15299799999866082	0.048591999999188284	2056.952728	2244.32986	'[]'
47	'Day-2023-2-15'	'mount'	0.1845829999997477	0.044918999999936204	2057.350745	2244.32986	'[]'
48	'Day-2023-2-16'	'mount'	0.16984999999970935	0.049000999999987366	2057.887976	2244.32986	'[]'
49	'Day-2023-2-17'	'mount'	0.14570000000028624	0.045275000000048163	2058.223159	2244.32986	'[]'
50	'Day-2023-2-18'	'mount'	0.14559499999950278	0.044228999999631924	2058.60953	2244.32986	'[]'
51	'Day-2023-2-19'	'mount'	0.15224899999985978	0.048642000000199914	2058.998922	2244.32986	'[]'
52	'Day-2023-2-20'	'mount'	0.14571200000045792	0.045155000000784644	2059.387887	2244.32986	'[]'
53	'Day-2023-2-21'	'mount'	0.14247499999985383	0.043754000000028275	2059.771879	2244.32986	'[]'
54	'Day-2023-2-22'	'mount'	0.1438309999998637	0.045141000000005783	2060.152327	2244.32986	'[]'
55	'Day-2023-2-23'	'mount'	0.14331099999981234	0.04474399999998096	2060.537651	2244.32986	'[]'
56	'Day-2023-2-24'	'mount'	0.14387099999930797	0.044995999999962811	2060.917471	2244.32986	'[]'
57	'Day-2023-2-25'	'mount'	0.14396000000076837	0.044150000000017244	2061.298193	2244.32986	'[]'

Рис. 10.14. Модульный тест выдает громадный объем информации об отрисовках

Обсуждение

Хотя мы создаем этот код для проверки производительности в модульном тесте Jest, он не является тестом в том же самом смысле, что и обычный тест функциональности, поскольку мы не выполняем никаких утверждений. Хотя утверждения могут быть полезными[2], нет смысла создавать тест производительности, который каждый раз показывает, что время исполнения некоторой операции меньше заданного значения. Результаты тестов производительности сильно зависят от среды. Если тест в среде разработки утверждает, что некая операция исполнится быстрее, чем за три секунды, не удивляйтесь, если он будет неуспешным на сервере интеграции, где эта же операция займет девять секунд.

Если же вы все-таки хотите отслеживать производительность автоматически, может быть целесообразным добавить регрессионное тестирование. Регрессионный тест записывает набор статистических данных по производительности в каком-

---

[2] Например, проверяя, что компонент находится в определенном состоянии, прежде чем выполнять какое-либо действие.

либо центральном хранилище вместе с идентификатором среды, создавшей эти данные. Это позволяет проверить, что дальнейшие прогоны теста ненамного медленнее предыдущих прогонов в той же среде.

По большому счету лучше предоставлять результаты тестирования производительности, а не выдавать подтверждение заданной вами желаемой оценки производительности.

Исходный код данного рецепта можно загрузить на веб-сайте GitHub по адресу **https://oreil.ly/0GfgA**.

10.4. Точное измерение времени

ЗАДАЧА

Когда в процессе тестирования мы дойдем до такой точки, где потребуется оптимизировать довольно низкоуровневый код JavaScript, возникнет вопрос: как в таком случае измерить производительность? Для этого можно, например, посредством функции `Date()` создавать временную метку в начале и конце кода JavaScript.

Листинг 10.8. Создание временной метки

```
const beforeDate = new Date()
for (let i = 0; i < 1000; i++) {}
const afterDate = new Date()
console.log(
   '1,000 loops took',
   afterDate.getTime() - beforeDate.getTime()
)
```

Каждую из этих дат можно преобразовать в миллисекунды, а затем отнять одно значение от другого, узнав таким образом длительность исполнения теста.

Это был настолько стандартный способ, что новым методам `time` и `timeEnd` был выделен консольный объект, чтобы сократить объем требуемого кода:

```
console.time('1,000 loops')
for (let i = 0; i < 1000; i++) {}
console.timeEnd('1,000 loops')
```

Функция `time` принимает метку в качестве параметра, и если вызвать функцию `timeEnd` с этой же меткой, она отобразит результаты в консоли. Но исполнение этого кода дает следующий результат:

```
1,000 loops: 0ms
```

Как видим, полученный результат некорректен. Приложения React редко содержат функции длительного исполнения, поэтому небольшие фрагменты кода JavaScript обычно необходимо оптимизировать только при их многократном вызове браузе-

ром. Например, может быть целесообразным оптимизировать код игры, который отрисовывает на экране анимацию. Оценить производительность небольших фрагментов кода трудно вследствие их чрезвычайно короткого времени исполнения, которое может не превышать одной миллисекунды. В подобных ситуациях измерять производительность с помощью объектов `Date` нельзя, поскольку они имеют разрешение величиной всего лишь в одну миллисекунду, хотя системные часы компьютера намного точнее.

Нам нужно какое-либо средство, которое позволило бы измерять промежутки времени длительностью меньше одной миллисекунды.

РЕШЕНИЕ

Для решения этой задачи мы воспользуемся функцией `performance.now()`. Вызов этой функции возвращает метку времени с высокой точностью, измеряемой в долях миллисекунды. Например, исполнение этой функции в консоли браузера Chrome дает результаты наподобие следующих:

```
> performance.now()
< 10131.62500000908
```

Как видим, здесь время измеряется по-другому, чем в объектах `Date` JavaScript. В JavaScript время отсчитывается начиная с 1 января 1970 г., а функция `performance.now()` измеряет время, начиная с момента загрузки текущей веб-страницы[3].

При попытке исполнить функцию `performance.now()` в браузере Firefox происходит интересная вещь:

```
> performance.now()
< 4194
```

По умолчанию Firefox возвращает только целую часть результата в миллисекундах, по сути, нивелируя большинство преимуществ данного способа. Это объясняется настройками безопасности данного браузера. Теоретически если JavaScript может измерять время исполнения крошечных фрагментов кода с высокой точностью, то это может создать сигнатуру браузера.

Чтобы включить высокое разрешение времени в Firefox, введите в строку адреса адрес `about:config`, в открывшемся списке настроек найдите параметр `privacy.reduceTimerPrecision` и присвойте ему значение `false`. После этого точность возвращаемых функцией `performance.now()` результатов увеличится:

```
performance.now()
151405.8
```

---

[3] При исполнении в Node функция `performance.now()` измеряет время, начиная с начала текущего процесса.

Обязательно установите для этого параметра его исходное значение `false`, если вы не желаете дать возможность сторонним лицам отслеживать вас.

Возвращаясь обратно к нашему примеру кода тестирования, мы можем измерить время исполнения циклов, как показано в листинге 10.9.

Листинг 10.9. Код для измерения времени исполнения циклов

```
const before0 = performance.now()
for (let i = 0; i < 1000; i++) {}
const after0 = performance.now()
console.log('1,000 loops took', after0 - before0)
const before1 = performance.now()
for (let i = 0; i < 100000; i++) {}
const after1 = performance.now()
console.log('100,000 loops took', after1 - before1)
```

Исполнение этого кода выдает следующее:

```
1,000 loops took 0.035767000000007804
100,000 loops took 1.6972319999999854
```

Эти результаты намного точнее и предоставляют больше информации о производительности соответствующего кода JavaScript. В данном случае можно видеть, что добавление в цикл дополнительных итераций не масштабирует время исполнения цикла линейным образом. Это наводит на мысль, что движок JavaScript начинает на лету оптимизировать код, как только он понимает, что все итерации цикла одинаковы.

Обсуждение

По сравнению с датами JavaScript функция `performance.now()` обладает несколькими преимуществами. Кроме дополнительной точности на эту функцию не воздействуют изменения часов, что является положительным аспектом, если вы решите добавить функциональность мониторинга производительности в код с длительным исполнением. Также она ведет отсчет времени от нуля, когда страница начинает загружаться, что полезно для оптимизации тайминга загрузки страницы.

Но на основе `performance.now()` не следует создавать высокоуровневые функции тайминга. Например, мы однажды создали простую функцию `timekeeper()` для генерирования кода JavaScript, чтобы немного упростить работу с функцией `performance.now()`(листинг 10.10).

Листинг 10.10. Код функции `timekeeper()`

```
function* timekeeper() {
   let now = 0
   while (true) yield -now + (now = performance.now())
}
```

Функция должна была устранить необходимость вычисления разницы между временем начала и окончания. Без этой функции такая задача решалась, как показано в листинге 10.11.

Листинг 10.11. Обычный код для вычисления разницы между начальным и конечным временем

```
const before0 = performance.now()
for (let i = 0; i < 1000; i++) {}
const after0 = performance.now()
console.log('1,000 loops took', after0 - before0)
const before1 = performance.now()
for (let i = 0; i < 100000; i++) {}
const after1 = performance.now()
console.log('100,000 loops took', after1 - before1)
```

А функция `timekeeper()` позволила бы выполнять эту задачу, как показано в листинге 10.12.

Листинг 10.12. Вычисление разницы между начальным и конечным временем при помощи функции `timekeeper()`

```
const t = timekeeper()
t.next()
for (let i = 0; i < 1000; i++) {}
console.log('1,000 loops took', t.next().value)
for (let i = 0; i < 100000; i++) {}
console.log('100,000 loops took', t.next().value)
```

Как видим, больше нет надобности в уродливых переменных `before` и `after`. После каждого вызова функции `t.nect().value` время бы обнулялось, устраняя необходимость в вычислениях.

Какая же с этим проблема? Заключение функции `performance.now()` внутри другой функции добавляет значительный объем времени к измерению, аннулируя ее точность:

```
1,000 loops took 0.05978800000002593
100,000 loops took 19.585223999999926
```

В данном случае, хотя исполнение 100 000 циклов занимает всего лишь 1,60 мс, наша функция возвращает результат, равный 19 мс.

Никогда не вставляйте функцию `performance.now()` в другую функцию, если вы хотите, чтобы она возвращала точные результаты.

Исходный код данного рецепта можно загрузить на веб-сайте GitHub по адресу **https://oreil.ly/0GfgA**.

10.5. Уменьшение размера приложений посредством разделения кода

ЗАДАЧА

Одно из самых отрицательных воздействий на производительность одностраничных приложений оказывает объем кода JavaScript, который нужно загрузить и исполнить. Мало того, что отрисовка кода JavaScript занимает время, так еще и увеличенная нагрузка на пропускную способность сети может значительно замедлить исполнение вашего приложения на устройствах, подключенных к мобильной сети.

Рассмотрим для примера приложение с синхронизированными маршрутами, которое мы использовали в *главе 2* (рис. 10.15).

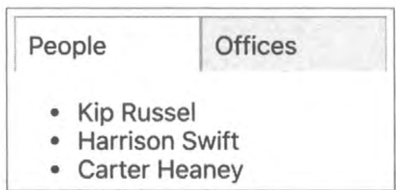


Рис. 10.15. Главное окно приложения с синхронизированными маршрутами

Хотя само приложение крошечного размера, оно содержит несколько довольно крупных пакетов кода JavaScript, как показано в листинге 10.13.

Листинг 10.13. Пакеты кода JavaSript для приложения с синхронизированными маршрутами

```
$ ls -l build/static/js
total 1336
-rw-r--r-- 1 davidg admin 161800 12:07 2.4db4d779.chunk.js
-rw-r--r-- 1 davidg admin 1290 12:07 2.4db4d779.chunk.js.LICENSE.txt
-rw-r--r-- 1 davidg admin 461100 12:07 2.4db4d779.chunk.js.map
-rw-r--r-- 1 davidg admin 4206 12:07 3.307a63d5.chunk.js
-rw-r--r-- 1 davidg admin 9268 12:07 3.307a63d5.chunk.js.map
-rw-r--r-- 1 davidg admin 3082 12:07 main.e8a3e1cb.chunk.js
-rw-r--r-- 1 davidg admin 6001 12:07 main.e8a3e1cb.chunk.js.map
-rw-r--r-- 1 davidg admin 2348 12:07 runtime-main.67df5f2e.js
-rw-r--r-- 1 davidg admin 12467 12:07 runtime-main.67df5f2e.js.map
$
```

Самый большой пакет (2.4db4d779.chunk.js) содержит основной код фреймворка React, а специфичный для приложения код содержится в небольшом файле main.e8a3e1cb.chunk.js. Это означает, что размер данного приложения настолько мал, насколько возможно для приложения React. Размер большинства приложений React значительно больше, часто до 1 Мб, что будет представлять значительную проблему для пользователей с медленным сетевым подключением.

Так что же можно сделать, чтобы уменьшить размер пакетов кода JavaScript в приложениях React?

РЕШЕНИЕ

Решением данной проблемы будет подход с *разделением кода*, который состоит в разбиении основного кода приложения на несколько пакетов меньшего размера. Тогда браузер сможет загрузить эти пакеты без особой спешки. Конкретный пакет будет загружаться только в случае надобности одного из содержащихся в нем компонентов.

Приложение, которое мы рассматриваем в качестве примера в этом разделе, конечно же, не требует разделения кода. Как и все корректировки для улучшения производительности, разделение кода следует предпринимать только в том случае, если это окажет существенное положительное воздействие на производительность приложения. Но мы выбрали это приложение для демонстрации разделения кода, поскольку так будет легче наблюдать действие этого процесса.

Для разделения кода приложений React предусмотрена функция `lazy`:

```
import { lazy } from 'react'
```

В качестве параметра эта функция принимает фабричную функцию, которая при вызове импортирует компонент. Функция `lazy` возвращает компонент-заглушку, который не делает ничего до тех пор, пока браузер не выполнит его отрисовку. Тогда компонент-заглушка исполняет фабричную функцию и динамически загружает пакет, содержащий настоящий компонент.

Чтобы разобраться, как это работает, рассмотрим компонент `About` из нашего демонстрационного приложения (листинг 10.14).

Листинг 10.14. Код компонента About

```
import { NavLink, Redirect, Route, Switch } from 'react-router-dom'
import People from './People'
import Offices from './Offices'
import './About.css'
const About = () => (
   <div className="About">
      <div className="About-tabs">
         <NavLink
            to="/about/people"
            className="About-tab"
            activeClassName="active"
         >
            People
         </NavLink>
         <NavLink
            to="/about/offices"
            className="About-tab"
```

```
            activeClassName="active"
         >
            Offices
         </NavLink>
      </div>
      <Switch>
         <Route path="/about/people">
            <People />
         </Route>
         <Route path="/about/offices">
            <Offices />
         </Route>
         <Redirect to="/about/people" />
      </Switch>
   </div>
)
export default About
```

Браузер выполняет отрисовку компонентов `People` и `Offices` только при посещении пользователем соответствующего маршрута. Если в настоящее время браузер находится на пути /about/people, то компонент `Offices` не будет отрисован. Значит, потенциально мы могли бы отложить загрузку данного компонента на более позднее время. Это можно реализовать при помощи функции `lazy`.

Заменим импортирование компонента `Offices` вызовом функции `lazy`:

```
//import Offices from "./Offices"
const Offices = lazy(() => import('./Offices'))
```

Объект, который теперь хранится в переменной `Offices`, будет выглядеть для остального кода приложения просто как любой другой компонент. Это так называемая *"ленивая" заглушка* (lazy placeholder). Она содержит ссылку на фабричную функцию, которую вызовет, когда браузер выполнит ее отрисовку.

Но если мы сейчас выполним обновление страницы, то браузер отобразит сообщение об ошибке (рис. 10.16).

```
Error: A React component suspended while rendering, but no fallback UI was specified.   ×

Add a <Suspense fallback=...> component higher in the tree to provide a loading indicator or
placeholder to display.

▸ 10 stack frames were collapsed.

Module.<anonymous>
src/index.js:7

    4 | import App from './App';
    5 | import reportWebVitals from './reportWebVitals';
    6 |
>   7 | ReactDOM.render(
    8 |   <React.StrictMode>
    9 |     <App />
   10 |   </React.StrictMode>,

View compiled
```

Рис. 10.16. Сообщение об ошибке при "ленивой" загрузке

Наш компонент-заглушка не будет ожидать загрузки настоящего компонента перед возвращением. Вместо этого он подставит какой-либо другой код HTML, пока ожидается завершение загрузки настоящего компонента.

Мы можем организовать этот интерфейс "загрузки", используя контейнер `Suspense`, как показано в листинге 10.15.

Листинг 10.15. Использование контейнера `Suspense`

```
import { lazy, Suspense } from 'react'
import { NavLink, Redirect, Route, Switch } from 'react-router-dom'
import People from './People'
// import Offices from './Offices'
import './About.css'
const Offices = lazy(() => import('./Offices'))
const About = () => (
   <div className="About">
      <div className="About-tabs">
         <NavLink
            to="/about/people"
            className="About-tab"
            activeClassName="active"
         >
            People
         </NavLink>
         <NavLink
            to="/about/offices"
            className="About-tab"
            activeClassName="active"
         >
            Offices
         </NavLink>
      </div>
      <Suspense fallback={<div>Loading...</div>}>
         <Switch>
            <Route path="/about/people">
               <People />
            </Route>
            <Route path="/about/offices">
               <Offices />
            </Route>
            <Redirect to="/about/people" />
         </Switch>
      </Suspense>
   </div>
)
export default About
```

Компонент-заглушка `lazy` ищет в своем контексте резервный компонент `fallback`, предоставляемый контейнером `Suspense`, и отображает его на странице, пока он ожидает загрузку дополнительного пакета кода JavaScript.

В данном случае мы выводим простое сообщение "Loading...", но можно использовать любой подстановочный интерфейс, чтобы создать у пользователя иллюзию, что новый компонент уже загрузился, прежде чем он в действительности загрузится. Например, этот же метод применяется на домашней странице YouTube. Пока загружается содержимое сайта, в браузере отображается набор блоков и прямоугольников вместо иконок видеоклипов, которые будут вскоре загружены (рис. 10.17). Для загрузки значков клипов часто требуются две-три секунды, но этот метод создает у пользователя впечатление, что они загружаются мгновенно.

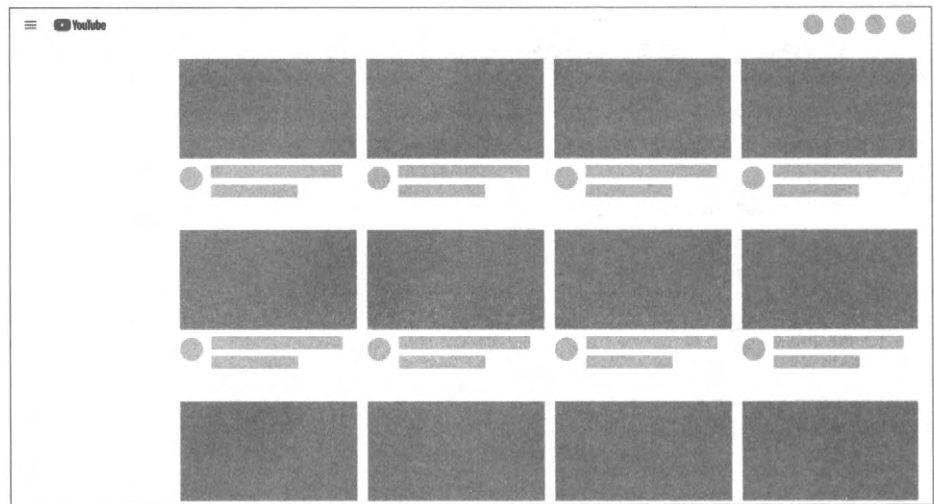


Рис. 10.17. Сайт YouTube отображает макет домашней страницы, пока загружаются рекомендуемые клипы

Если сейчас обновить страницу нашего демонстрационного приложения, оно снова начнет отображаться нормально, как показано на рис. 10.18.

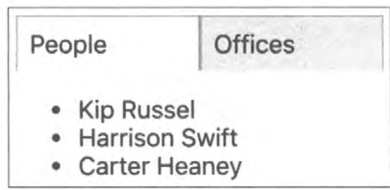


Рис. 10.18. Добавление компонента `Suspense` устраняет ошибку загрузки

Тем временем за кулисами сервер для разработки Webpack выделит код компонента `Offices` в отдельный пакет кода JavaScript.

Он также разделит код на отдельные пакеты при создании сборки. Для этого сервер Webpack использует метод *встряхивания дерева* (tree shaking), чтобы определить,

те компоненты, которые можно без проблем разместить в определенные пакеты JavaScript.

Встряхивание дерева — это процесс, который рекурсивно анализирует, какие файлы с кодом импортируются другими файлами, начиная с какого-либо исходного файла, например index.js. Это позволяет серверу Webpack избежать добавления в пакет кода, который никогда не импортируется никаким другим кодом. Вызовы функции `React.lazy` не отслеживаются данным процессом, поэтому код с "ленивой" загрузкой не включается в начальный пакет JavaScript. Вместо этого сервер Webpack исполняет отдельный процесс встряхивания дерева для каждого файла с "ленивой" загрузкой, в результате чего в рабочем приложении создается множество пакетов кода небольшого размера.

Если теперь создать новую сборку, а затем проверить папку js сборки, то мы увидим, что она содержит несколько новых файлов, как показано в листинге 10.16.

Листинг 10.16. Список файлов в новой сборке

```
$ yarn build
...Builds code
$ ls -l build/static/js
total 1352
-rw-r--r-- 1 davidg admin 628 12:09 0.a30b3768.chunk.js
-rw-r--r-- 1 davidg admin 599 12:09 0.a30b3768.chunk.js.map
-rw-r--r-- 1 davidg admin 161801 12:09 3.f7664178.chunk.js
-rw-r--r-- 1 davidg admin 1290 12:09 3.f7664178.chunk.js.LICENSE.txt
-rw-r--r-- 1 davidg admin 461100 12:09 3.f7664178.chunk.js.map
-rw-r--r-- 1 davidg admin 4206 12:09 4.a74be2bf.chunk.js
-rw-r--r-- 1 davidg admin 9268 12:09 4.a74be2bf.chunk.js.map
-rw-r--r-- 1 davidg admin 3095 12:09 main.e4de2e45.chunk.js
-rw-r--r-- 1 davidg admin 6089 12:09 main.e4de2e45.chunk.js.map
-rw-r--r-- 1 davidg admin 2361 12:09 runtime-main.9df06006.js
-rw-r--r-- 1 davidg admin 12496 12:09 runtime-main.9df06006.js.map
```

Поскольку размер нашего демонстрационного приложения невелик, вряд ли описанная оптимизация как-либо повлияет на производительность, но давайте проверим в любом случае.

Проверить производительность загрузки легче всего при помощи инструмента Lighthouse браузера Chrome. На рис. 10.19 показаны данные измерения производительности оригинальной версии этого приложения.

Оптимизация приложения добавлением в него возможности "ленивой" загрузки слегка повышает производительность, в основном за счет уменьшения времени на завершение FCP (рис. 10.20).

Полученное улучшение производительности нельзя назвать существенным, но тем не менее оно показывает, что "ленивая" загрузка может быть полезной даже для крошечных приложений.

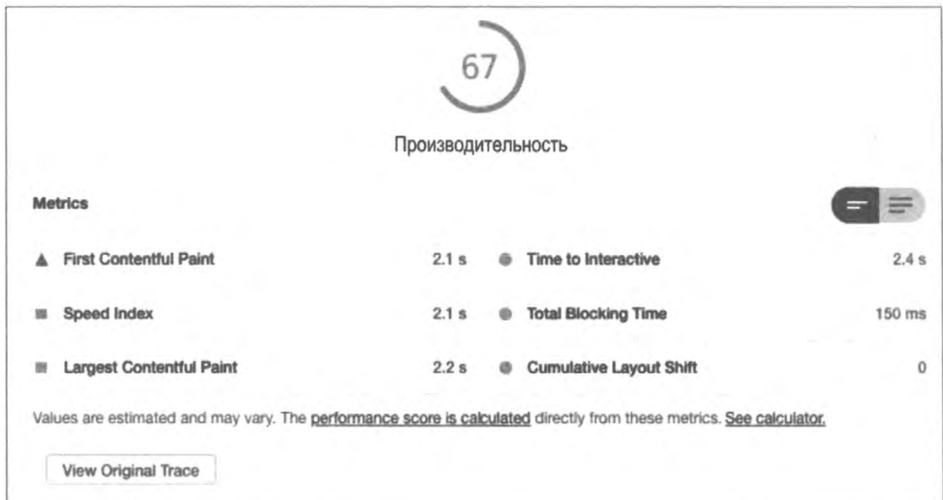


Рис. 10.19. Данные производительности приложения без разделения кода

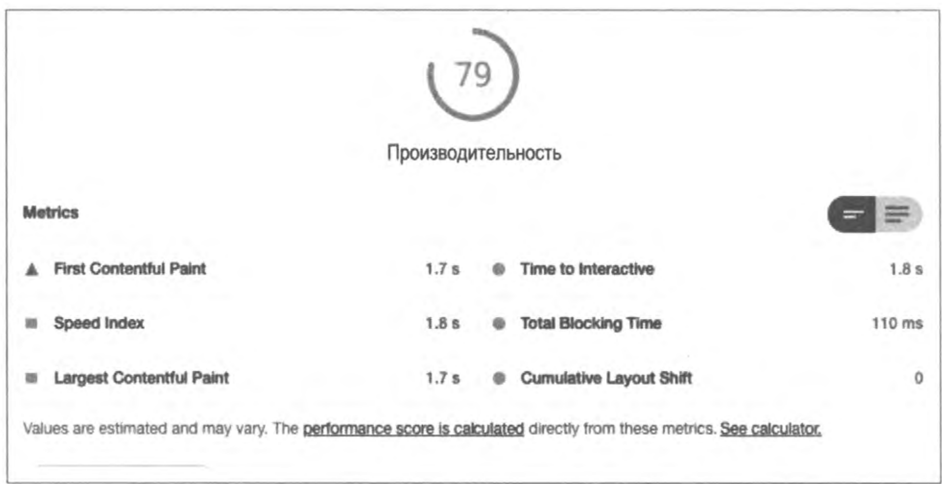


Рис. 10.20. Данные производительности приложения с разделением кода

Обсуждение

Все виды оптимизации имеют свою цену, но реализация разделения кода требует минимальных усилий, и мы используем эту оптимизацию наиболее часто. Данный метод нередко улучшает такие важные веб-показатели, как FCP и TTI. Чрезмерно увлекаться этим не стоит, поскольку фреймворку требуется выполнять дополнительную работу по загрузке и оценке каждого сценария. Но для большинства достаточно крупных приложений разделение кода немедленно даст определенный положительный эффект.

Разделение кода часто лучше всего выполнять на уровне маршрутов. Маршруты управляют видимостью компонентов и поэтому будут хорошим местом для отделения кода, который нужно загрузить сейчас, от кода, который необходимо загрузить позже. К тому же если пользователь создаст закладку в каком-либо месте вашего приложения, то при переходе по этой закладке он загрузит только код, требуемый для данного места.

Исходный код данного рецепта можно загрузить на веб-сайте GitHub по адресу **https://oreil.ly/0GfgA**.

10.6. Объединение сетевых обещаний

ЗАДАЧА

Многие приложения React осуществляют асинхронные сетевые вызовы, ожидание ответа на которые сильно тормозит работу приложения. В течение этих вызовов приложение, скорее всего, не выполняет какой-либо существенной работы, поэтому оно не занято, а просто ожидает.

С течением времени уровень сложности клиентских приложений повысился, а уровень сложности API-серверов — снизился. В случае бессерверных приложений API-интерфейсы серверов стали настолько стандартными, что не требуют никакого специализированного кода, что ведет к повышению числа вызовов API-интерфейса, осуществляемых клиентским кодом[4].

Рассмотрим соответствующий пример. Имеется приложение, которое считывает данные нескольких людей из API-интерфейса сервера бэкенда. Сервер содержит конечную точку, которая при получении от браузера запроса `GET` по адресу /people/1234 возвратит данные человека с идентификатором `1234`. Для формирования этих запросов разработчик создал хук, код которого показан в листинге 10.17.

Листинг 10.17. Код для создания запросов к конечной точке

```
import { useEffect, useState } from 'react'
import { get } from './fakeios'
const usePeopleSlow = (...ids) => {
   const [people, setPeople] = useState([])
   useEffect(() => {
      let didCancel = false
      ;(async () => {
         const result = []
         for (let i = 0; i < ids.length; i++) {
            const id = ids[i]
```

[4] Исключение составляет служба GraphQL. В этой среде клиент может отправить серверу бэкенда сложный запрос, и стандартизированный интерпретатор запросов соберет вместе на сервере результаты низкоуровневых запросов. Среда исполнения GraphQL позволяет получить быстрые сетевые ответы без необходимости выполнять настройку клиента.

```
            result.push(await get('/people/' + id))
         }
         if (!didCancel) {
            setPeople(result)
         }
      })()
      return () => {
         didCancel = true
      }
      // eslint-disable-next-line react-hooks/exhaustive-deps
   }, [...ids])
   return people
}
export default usePeopleSlow
```

Этот хук вызывается следующим образом:

```
const peopleSlow = usePeopleSlow(1, 2, 3, 4)
```

Данный код отправляет серверу вызов для каждого значения идентификатора. Перед тем как сохранить ответы в массиве, код ожидает ответы на все запросы. Таким образом, если конечной точке API-интерфейса требуется 5 секунд для выдачи ответа, то, чтобы возвратить все данные, хуку `usePeopleSlow` потребуется 20 секунд.

Можно ли каким-либо образом ускорить этот процесс?

РЕШЕНИЕ

Для решения данной задачи мы объединим асинхронные обещания, чтобы одновременно в процессе подачи было несколько запросов к API-интерфейсу.

Большинство библиотек для работы с асинхронными запросами возвращают обещания. Если ожидать обработки обещания, то оно возвратит полезную нагрузку ответа. Но в нашем примере с кодом хука `usePeopleSlow` ожидание обработки этих обещаний осуществляется в последовательности, как показано в листинге 10.18.

Листинг 10.18. Ожидание обработки обещаний

```
const result = []
for (let i = 0; i < ids.length; i++) {
   const id = ids[i]
   result.push(await get('/people/' + id))
}
```

Запрос для второго человека не отправляется, пока не будет получен ответ на запрос для первого человека. Это и является причиной, почему 5-секундная задержка превращается в 20-секундное время отклика при считывании данных для четырех людей.

Данную задачу можно решить другим способом. В частности, запросы можно отправлять, не ожидая ответа, чтобы все они находились в процессе обработки одновременно. Затем нам нужно дождаться всех ответов, и, получив последний из них, мы можем возвратить данные из хука.

Реализовать параллельную отправку запросов можно при помощи функции JavaScript `Promise.all`.

Эта функция принимает в качестве параметров список обещаний и объединяет их в одно обещание. Таким образом, мы можем объединить несколько вызовов `get()`, как показано в листинге 10.19.

Листинг 10.19. Объединение нескольких вызовов `get()`

```
const [res1, res2, res3] = await Promise.all(
   get('/people/1'),
   get('/people/2'),
   get('/people/3')
)
```

Функция `Promise.all` объединяет не только обещания, но также и результаты. Если дождаться массива обещаний с функцией `Promise.all`, мы получим массив, содержащий все обещания.

Теперь мы можем модифицировать хук `usePeopleSlow`, используя в нем функцию `Promise.all`. Соответствующий код приведен в листинге 10.20.

Листинг 10.20. Версия хука `usePeopleFast` с использованием функции `Promise.all`

```
import { useEffect, useState } from 'react'
import { get } from './fakeios'
const usePeopleFast = (...ids) => {
   const [people, setPeople] = useState([])

   useEffect(() => {
      let didCancel = false
      ;(async () => {
         const result = await Promise.all(
            ids.map((id) => get('/people/' + id))
         )
         if (!didCancel) {
            setPeople(result)
         }
      })()
      return () => {
         didCancel = true
      }
```

```
    // eslint-disable-next-line react-hooks/exhaustive-deps
  }, [...ids])

  return people
}
export default usePeopleFast
```

Ключевыми в этом коде являются следующие три строки:

```
const result = await Promise.all(
  ids.map((id) => get('/people/' + id))
)
```

Сопоставляя (метод `ids.map`) идентификаторы элементам массива обещаний, возвращенных сетевыми запросами, мы можем дождаться результата функции `Promise.all` и получить массив со всеми ответами.

Если измерить время исполнения обоих хуков, то увидим, что хук `usePeopleFast` считывает данные четырех пользователей чуть больше, чем за пять секунд. Фактически мы исполнили пять запросов за время, требуемое для исполнения одного. В табл. 10.2 для сравнения приведено время исполнения каждой версии хука `usePeople`.

Таблица 10.2. *Время исполнения каждой версии хука* `usePeople`

Версия хука	Время исполнения, мс
usePeopleSlow	20011,224999994738
usePeopleFast	5000,99999998929

Обсуждение

Данный подход значительно улучшит производительность сетевых запросов, даже при наличии множественных независимых асинхронных запросов. Если отправлять большое количество параллельных запросов, то браузер, сетевая плата или сервер могут начать ставить их в очередь. Но этот метод тем не менее позволяет получить ответ быстрее, нежели последовательность отдельных запросов.

Параллельные запросы усиливают нагрузку на сервер, но это вряд ли будет сколько-либо большой проблемой. Во-первых, как уже упоминалось, серверы часто ставят запросы в очередь, когда они заняты. Во-вторых, сервер будет одинаково выполнять общий объем работы. Мы просто концентрируем эту работу в более короткий период времени.

Исходный код данного рецепта можно загрузить на веб-сайте GitHub по адресу **https://oreil.ly/0GfgA**.

10.7. Отрисовка на стороне сервера

ЗАДАЧА

Одностраничные приложения наполняют веб-страницы различными возможностями, делая их похожими на настольные приложения. Работа с приложением наподобие Google Docs почти неотличима от использования текстового редактора на персональном компьютере.

Но все имеет свою цену. Одна из основных проблем с производительностью одностраничных приложений состоит в том, что прежде чем браузер может создать интерфейс, ему нужно загрузить большой объем кода JavaScript. Тело кода HTML приложений, созданных при помощи такого инструмента, как `create-react-app`, содержит лишь пустой раздел `<div>` с идентификатором `root`:

```
<div id="root"></div>
```

И лишь этот пустой раздел `<div>` будет доступен браузеру для отображения, пока движок JavaScript не загрузит сопутствующий код JavaScript, исполнит его и обновит модель DOM.

Даже если уменьшить размер загружаемого кода, разделив его на отдельные пакеты, и учесть, что браузер кеширует код JavaScript, чтение кода и организация интерфейса могут тем не менее занять пару секунд.

Создание всего интерфейса полностью на основе кода JavaScript означает, что одностраничные приложения страдают двумя основными проблемами. Во-первых, и наиболее важно то, что восприятие пользователем может ухудшиться, особенно в случае больших приложений React. Во-вторых, поисковая оптимизация (SEO) приложения будет плохой. При сканировании сайтов роботы поисковых систем часто не ждут, пока JavaScript выполнит отрисовку интерфейса, а загружают базовый HTML-код страницы и индексируют его содержимое. Для многих бизнес-приложений это, может, не слишком важно. Но для, например, сайта интернет-магазина скорее всего будет желательным, чтобы было проиндексировано как можно больше страниц, чтобы увеличить число посещений.

Поэтому будет полезным, если вместо отображения пустого раздела `<div>`, пока загружается код HTML, можно было бы включить в страницу приложения начальный код HTML перед тем, как браузер загрузит и исполнит код JavaScript-приложения.

РЕШЕНИЕ

Для решения данной задачи мы рассмотрим, как использовать отрисовку на стороне сервера, чтобы заменить пустой раздел `<div>` страницы React заранее отрисованным кодом HTML. Это возможно благодаря механизму взаимодействия React с моделью DOM веб-страницы.

При отрисовке компонента в React вместо непосредственного обновления модели DOM исполняется фрагмент кода, приведенный в листинге 10.21.

Листинг 10.21. Код для обновления виртуальной модели DOM

```
ReactDOM.render(
  <React.StrictMode>
    <App />
  </React.StrictMode>,
  document.getElementById('root')
)
```

Метод `render` обновляет виртуальную модель DOM, которая периодически синхронизируется с настоящими элементами HTML-страницы. Среда React выполняет это обновление рациональным способом, обновляя в реальной модели DOM только те элементы, которые не совпадают с элементами в виртуальной модели DOM.

Отрисовка на стороне сервера выполняется не в виртуальную модель DOM среды React, а в строку. Когда браузер запросит у сервера HTML-страницу, мы выполним отрисовку версии содержимого React в строку, затем вставим эту строку в код HTML и лишь тогда возвратим код браузеру. Это означает, что браузер немедленно отобразит содержимое HTML-страницы, до того как он начнет загружать код JavaScript-приложения. Код для реализации этого на стороне сервера может выглядеть так, как показано в листинге 10.22.

Листинг 10.22. Код для предварительной отрисовки на стороне сервера

```
let indexHTML = <contents of index.html>
const app = <render App to string>
indexHTML = indexHTML.replace(
  '<div id="root"></div>',
  `<div id="app">${app}</div>`
)
res.contentType('text/html')
res.status(200)
return res.send(indexHTML)
```

Чтобы разобраться с этим подходом более подробно, создадим приложение `create-react-app`.

Отрисовку на стороне сервера поддерживают многие инструменты и фреймворки React, но средство `create-react-app` ***не входит*** в их число. Поэтому рассмотрим, как преобразовать приложение `create-react-app` в версию с поддержкой отрисовки на сервере (SSR), чтобы понять все шаги, необходимые для реализации отрисовки на сервере в React:

```
$ npx create-react-app ssrapp
```

Нам нужно создать сервер, чтобы разместить на нем код отрисовки. Начнем с создания папки для хранения серверного кода:

```
$ mkdir server
```

Для создания сервера мы используем средство Express. Наш сервер код будет выполнять отрисовку компонентов приложения.

Нам потребуется несколько дополнительных библиотек, чтобы помочь с загрузкой компонентов React. В основной папке приложения (не в подпапке для сервера) выполните следующую команду установки:

```
$ npm install --save-dev ignore-styles url-loader @babel/register
```

Средство `create-react-app` создает код, использующий большое количество современных возможностей JavaScript, которые недоступны в установочном варианте среды. Поэтому первое, что нам нужно сделать в нашем серверном коде, — разрешить эти возможности JavaScript, чтобы сервер мог исполнять наши компоненты React. Для этого в созданной нами ранее папке для сервера создайте файл index.js и вставьте в него код, приведенный в листинге 10.23.

Листинг 10.23. Код файла index.js

```
require('ignore-styles')
require('url-loader')
require('file-loader')
require('regenerator-runtime/runtime')
require('@babel/register')({
   ignore: [/(node_modules)/],
   presets: [
      '@babel/preset-env',
      [
         '@babel/preset-react',
         {
            runtime: 'automatic',
         },
      ],
   ],
plugins: [],
})
require('./ssr')
```

Этот файл содержит настройки возможностей языка JavaScript, которые мы будем использовать в серверном коде. В частности, мы загружаем подключаемый модуль `preset-react` компилятора Babel, который устанавливается автоматически в каждом приложении `create-react-app`. В конце этого сценария загружается файл ssr.js, в который мы поместим наш основной серверный код.

Далее в папке сервера создайте файл ssr.js и вставьте в него код, приведенный в листинге 10.24.

Листинг 10.24. Серверный код для файла ssr.js

```
import express from 'express'
import fs from 'fs'
```

```
import path from 'path'
const server = express()
server.get(
  /.(js|css|map|ico|svg|png)$/,
  express.static(path.resolve(__dirname, '../build'))
)

server.use('*', async (req, res) => {
  let indexHTML = fs.readFileSync(
    path.resolve(__dirname, '../build/index.html'),
    {
      encoding: 'utf8',
    }
  )
  res.contentType('text/html')
  res.status(200)
  return res.send(indexHTML)
})
server.listen(8000, () => {
  console.log(`Launched at http://localhost:8000!`)
})
```

Наш пользовательский сервер будет работать подобно серверу для разработки, содержащему средство `create-react-app`. Он создает веб-сервер следующей строкой кода:

```
const server = express()
```

При получении сервером запроса на код JavaScript, таблицу стилей или изображение он будет искать соответствующий файл в папке build (листинг 10.25). В этой папке средство `create-react-app` сохраняет созданную им развертываемую версию приложения.

Листинг 10.25. Поиск сервером запрошенного материала в папке *build*

```
server.get(
  /.(js|css|map|ico|svg|png)$/,
  express.static(path.resolve(__dirname, '../build'))
)
```

На запросы материала любого другого типа сервер возвращает содержимое файла build/index.html:

```
server.use('*', async (req, res) => {
  ...
})
```

Наконец, сервер запускается на порту 8000:

```
server.listen(8000, () => {
  console.log(`Launched at http://localhost:8000!`)
})
```

Но прежде чем запускать сервер, нужно выполнить сборку нашего приложения, задав следующую команду:

```
$ yarn run build
```

В процессе сборки приложения в папке build создаются все статические файлы, которые будут нужны нашему серверу. Вот теперь сервер можно запустить на исполнение:

```
$ node server
Launched at http://localhost:8000!
```

Запустив сервер, открываем браузер по адресу **http://localhost:8000**. В браузере должно отобразиться наше приложение React (рис. 10.21).

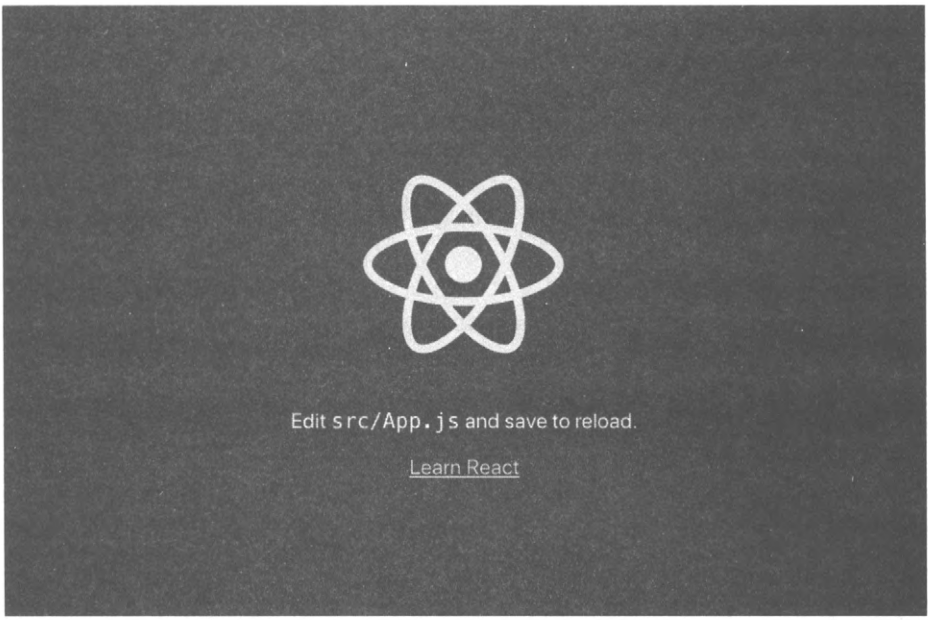


Рис. 10.21. Приложение React, раздаваемое созданным нами сервером

Пока что все идет по плану. Но в действительности мы не выполняем никакой отрисовки на стороне сервера. Для этого нам нужно загрузить немного кода React для загрузки и отрисовки компонента `App` (листинг 10.26).

Листинг 10.26. Код для загрузки и отрисовки компонента `App`

```
import express from 'express'
import fs from 'fs'
```

```
import path from 'path'
import { renderToString } from 'react-dom/server'
import App from '../src/App'
const server = express()
server.get(
   /.(js|css|map|ico|svg|png)$/,
   express.static(path.resolve(__dirname, '../build'))
)
server.use('*', async (req, res) => {
   let indexHTML = fs.readFileSync(
      path.resolve(__dirname, '../build/index.html'),
      {
         encoding: 'utf8',
      }
   )

   const app = renderToString(<App />)

   indexHTML = indexHTML.replace(
      '<div id="root"></div>',
      `<div id="app">${app}</div>`
   )
   res.contentType('text/html')
   res.status(200)
return res.send(indexHTML)
})
server.listen(8000, () => {
   console.log(`Launched at http://localhost:8000!`)
})
```

В коде в листинге 10.26 используется функция `renderToString` из библиотеки `react-dom/server` React. Эта функция делает то, что можно ожидать по ее названию. Вместо отрисовки компонента `App` в виртуальную модель DOM она выполняет его отрисовку в строку. Таким образом, мы можем заменить пустой раздел `<div>` в файле index.html кодом HTML, сгенерированным из компонента `App`. Если теперь перезапустить сервер и перезагрузить веб-браузер, мы увидим, что, хотя приложение и продолжает работать, функционирует оно не совсем правильно (рис. 10.22).

Вместо изображения вращающегося логотипа React мы видим его искаженную версию. Чтобы понять причину этого, взглянем на возвращенный сервером сгенерированный на нем код HTML, который приведен в листинге 10.27.

Листинг 10.27. Возвращенный сервером код HTML

```
<div id="app">
   <div class="App" data-reactroot="">
      <header class="App-header">
```

```
        <img src="[object Object]" class="App-logo" alt="logo"/>
        <p>Edit <code>src/App.js</code> and save to reload.</p>
        <a class="App-link" href="https://reactjs.org"
           target="_blank" rel="noopener noreferrer">
           Learn React
        </a>
     </header>
   </div>
</div>
```

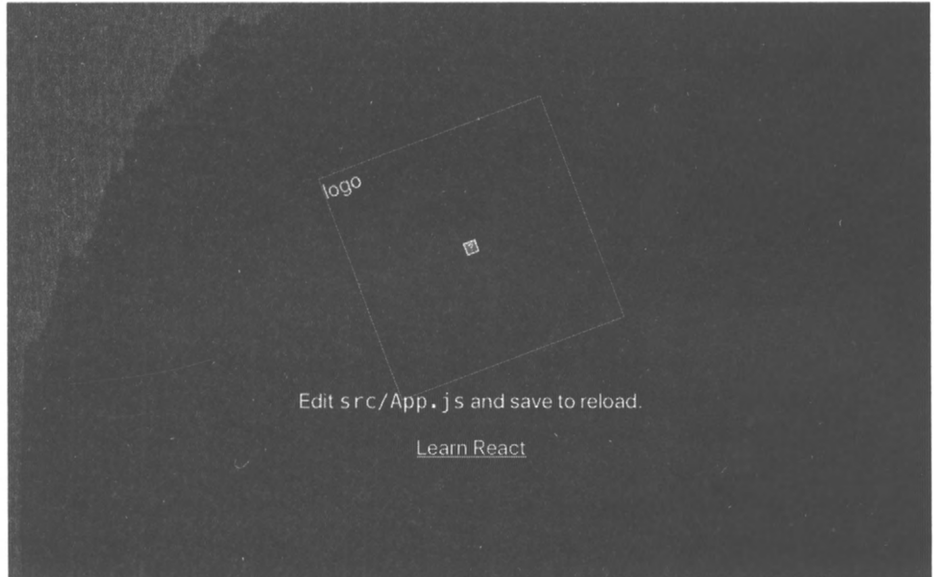


Рис. 10.22. Приложение React отображает искаженное изображение SVG

Обратим внимание на то, что с элементом `img` произошло что-то странное. Вместо того, чтобы выполнять отрисовку изображения SVG, он пытается загрузить URL `[object Object]`. Что здесь происходит?

В коде React логотип загружается следующим образом:

```
import logo from './logo.svg'
   ...
<img src={logo} className="App-logo" alt="logo" />
```

Этот код полагается на какую-то конфигурацию Webpack из средства `create-react-app`. При обращении к приложению через сервер для разработки сборщик модулей Webpack заменяет любые импортированные файлы SVG сгенерированными React-компонентами, содержащими необработанное содержимое SVG, используя для этого библиотеку `svgr`. Библиотека `svgr` позволяет загружать изображения SVG, как любые другие компоненты React. Таким образом, мы можем импортировать эти изображения как простой файл *.js.

Но созданный нами сервер не содержит такой конфигурации Webpack. Вместо того чтобы заморачиваться с настройкой Webpack на сервере, мы можем скопировать файл логотипа logo.svg в папку public, а затем заменить код в компоненте App кодом из листинга 10.28.

Листинг 10.28. Новый код компонента App

```
// import logo from './logo.svg'
import './App.css'
function App() {
   return (
      <div className="App">
         <header className="App-header">
            <img src="/logo.svg" className="App-logo" alt="logo" />
            <p>
               Edit <code>src/App.js</code> and save to reload.
            </p>
            <a
               className="App-link"
               href="https://reactjs.org"
               target="_blank"
               rel="noopener noreferrer"
            >
               Learn React
            </a>
         </header>
   </div>
   )
}
export default App
```

Далее снова выполним сборку приложения и перезапустим сервер:

```
$ yarn build
$ node server
```

Теперь наше одностраничное приложение должно отображаться правильно, как показано на рис. 10.23.

Но нам нужно выполнить еще один шаг. Отрисовка одностраничной версии нашего приложения осуществляется кодом из файла src/index.js (листинг 10.29).

Листинг 10.29. Код файла src/index/js для отрисовки одностраничного приложения

```
ReactDOM.render(
   <React.StrictMode>
      <App />
   </React.StrictMode>,
   document.getElementById('root')
)
```

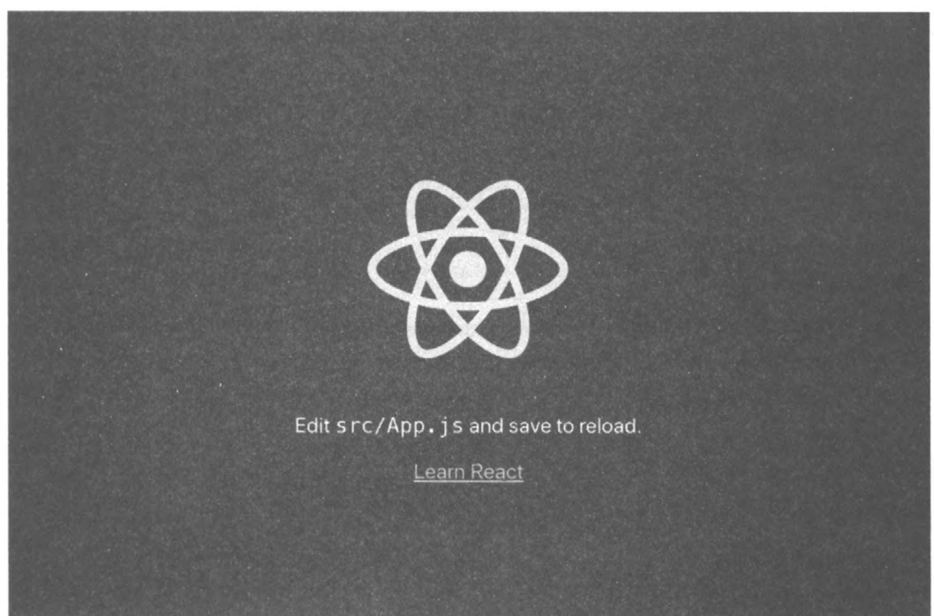


Рис. 10.23. Теперь приложение отображает изображение SVG правильным образом

Этот код также будет исполняться, даже когда мы обращаемся к приложению через сервер для отрисовки на стороне сервера. В таком случае браузер загружает предварительно отрисованную версию веб-страницы, а затем загрузит код JavaScript для одностраничного приложения. При исполнении кода одностраничного приложения будет исполнен код из файла index.js, приведенный в листинге 10.29. Браузеру все еще нужно загружать и исполнять код JavaScript, чтобы сделать интерфейс интерактивным. Метод `ReactDOM.render` может заменять все наше предварительно отрисованное HTML-содержимое, даже когда в этом нет надобности. Поэтому после замены вызова этого метода вызовом метода `ReactDOM.hydrate` HTML-содержимое в модели DOM будет меняться только тогда, когда оно отличается от HTML-содержимого в виртуальной модели DOM. Содержимое нашей отрисованной на стороне сервера страницы должно быть таким же, как и содержимое виртуальной модели DOM. В результате метод `hydrate` не будет обновлять элементы страницы, а просто присоединит к ней набор слушателей событий, чтобы сделать страницу интерактивной.

Таким образом, у нас теперь есть приложение с отрисовкой на стороне сервера. Но загружается ли такое приложение сколько-либо быстрее?

Самый простой способ проверить время загрузки страницы — выполнить для нее оценку производительности Lighthouse в браузере. Как упоминалось ранее, средство Lighthouse проверяет производительность, доступность и целую кучу других характеристик. Оно сможет предоставить нам показатели, при помощи которых мы в состоянии сравнить производительность наших двух версий приложения.

Когда мы выполнили эту проверку на нашем инструментальном ноутбуке, обращаясь к обычному серверу React для разработки, встроенному в средство `create-react-`

app, мы получили оценку производительности величиной 91 из 100 возможных и время FCP, равное 1,2 секунды (рис. 10.24).

Это неплохая оценка производительности. Но, опять же, мы исполняем небольшое приложение React.

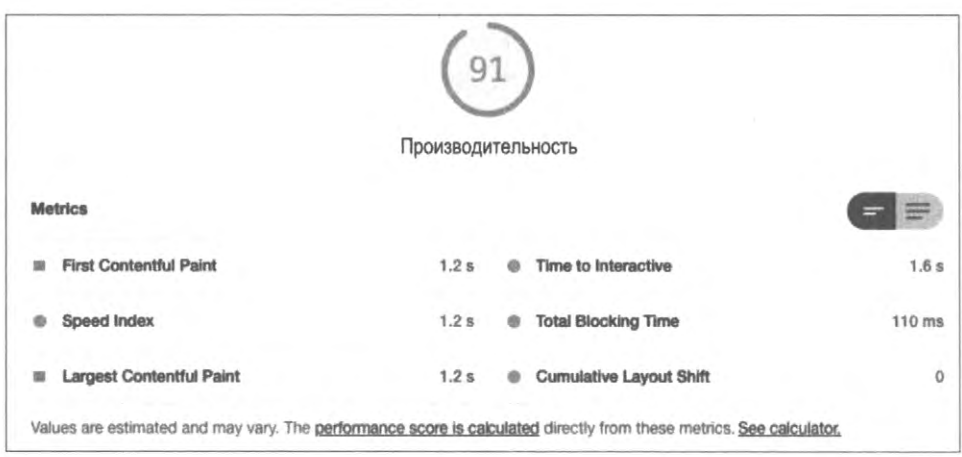


Рис. 10.24. Базовая производительность приложения без отрисовки на стороне сервера

А какие будут результаты проверки версии приложения с отрисовкой на стороне сервера? Ведь серверу все же придется потратить некоторое время для отрисовки кода React. Будет ли приложение исполняться сколько-либо быстрее? Результаты оценки производительности версии приложения с отрисовкой на стороне сервера показаны на рис. 10.25.

99

Производительность

Metrics

First Contentful Paint 0.6 s
Time to Interactive 0.6 s
Speed Index 0.6 s
Total Blocking Time 0 ms
Largest Contentful Paint 0.8 s
Cumulative Layout Shift 0

Values are estimated and may vary. The performance score is calculated directly from these metrics. See calculator.

Рис. 10.25. Базовая производительность приложения с отрисовкой на стороне сервера

Общая оценка повысилась с 99 до 100. Значение времени для FCP уменьшилось до 0,6 секунды, т. е. вдвое по сравнению с исходной версией приложения. Кроме того, если при исполнении исходной версии приложения в браузере многократно нажи-

мать кнопку обновления страницы, то часто перед отображением обновленного содержимого будет кратковременно отображаться пустое окно белого цвета. Это происходит по той причине, что загруженное HTML-содержимое состоит всего лишь из пустого раздела `<div>`, который браузер и отображает как белую страницу перед тем, как JavaScript сможет отобразить приложение.

Сравним это поведение с поведением версии приложения с отрисовкой на стороне сервера. В этом случае повторные нажатия кнопки обновления вызывают только возврат в исходное положение изображения логотипа, и не наблюдается никаких белых экранов.

Хотя на сервере все же происходит процесс отрисовки, который для строчной версии HTML-содержимого требует меньше времени, чем для отрисовки этого же набора элементов DOM.

Обсуждение

В этом разделе мы рассмотрели основные положения, как можно организовать отрисовку приложения на стороне сервера. Подробности реализации для вашего приложения будут, скорее всего во многом другими, в зависимости от используемых в нем дополнительных библиотек.

Например, в большинстве приложений React применяется маршрутизация. При работе с пакетом `react-router` в серверный код нужно будет добавить дополнительный код, чтобы иметь возможность выполнять отрисовку разных компонентов, в зависимости от запрошенного браузером пути. Например, в листинге 10.30 показано возможное использование компонента `StaticRouter` из пакета `react-router`.

Листинг 10.30. Возможное использование компонента StaticRouter

```
import { StaticRouter } from 'react-router-dom'
...
const app = renderToString(
   <StaticRouter location={req.originalUrl} context={{}}>
      <App />
   </StaticRouter>
)
```

Компонент `StaticRouter` выполняет отрисовку своих дочерних элементов для одного конкретного маршрута. В данном случае мы задали маршрут `originalURL` из запроса браузера. Если браузер запросит маршрут /person/1234, то компонент `StaticRouter` выполнит отрисовку компонента `App` для этого маршрута.

Обратите внимание на то, что компонент `StaticRouter` также подойдет для передачи любого дополнительного содержимого остальному приложению. Для этой цели также можно было бы использовать контекст.

В случае применения в разрабатываемом приложении разделения кода с помощью функции `React.lazy` следует иметь в виду, что этот метод не будет работать на сто-

роне сервера. Но, к счастью, существует способ обойти это ограничение. В частности, вместо функции `React.lazy` можно использовать библиотеку Loadable Components, которая обеспечивает ту же функциональность, но может исполняться также и на стороне сервера. Таким образом, эта библиотека предоставляет все достоинства отрисовки на стороне сервера вместе с преимуществами разделения кода.

Но, как и с любым типом оптимизации, за отрисовку на стороне сервера нужно расплачиваться усложнением кода и дополнительной нагрузкой на сервер. Версию одностраничного приложения со статическим кодом можно развернуть на любом веб-сервере. Но приложения с отрисовкой на стороне сервера такой возможностью не обладают. Для них нужен сервер JavaScript, что может повысить стоимость ваших хостинговых услуг.

Кроме того, если вы с самого начала знаете, что будете выполнять отрисовку на стороне сервера, вам, наверное, следует рассмотреть возможность использования такого инструмента, как Razzle или Next.js, и применить этот подход с первого дня разработки своего приложения.

Наконец, для повышения производительности веб-страницы существуют и альтернативные подходы, без необходимости прибегать к отрисовке на стороне сервера. Например, можно задействовать фреймворк Gatsby. Это средство может выполнять предварительную отрисовку страницу в процессе сборки приложения, предоставляя многие из преимуществ отрисовки на стороне сервера без необходимости создания серверного кода.

Исходный код данного рецепта можно загрузить на веб-сайте GitHub по адресу **https://oreil.ly/0GfgA**.

10.8. Использование основных показателей веб-производительности

ЗАДАЧА

Намного важнее иметь рабочий и легкочитаемый код, нежели высокооптимизированный код. Как упоминалось ранее, оптимизация всегда имеет свою цену.

Но если в приложении возникают заметные проблемы с производительностью, чрезвычайно важно обратить на них внимание и исправить их как можно быстрее. Большая часть интернет-торговли полагается на покупки случайными посетителями веб-магазина. Но если посетитель вашего веб-сайта должен ожидать его загрузки слишком долго, то он может уйти и больше никогда не вернуться.

Разработчики часто отслеживают производительность серверов с помощью средств отслеживания (называющихся маяками — beacons), встроенных в код. В случае наличия проблемы с производительностью маяк выдает сигнал тревоги, и разработчик может исправить эту проблему, прежде чем она окажет отрицательное воздействие на большое количество посетителей сайта.

Но как можно встроить отслеживающий маяк в клиентский код?

РЕШЕНИЕ

Для решения этой задачи мы рассмотрим, как отслеживать основные показатели производительности веб-приложений. Мы уже упоминали показатели производительности в *разделе 10.1*. Это небольшой набор показателей производительности, измеряющих наиболее важные параметры приложения. Например, показатель CLS измеряет величину смещения части страницы в процессе загрузки.

Для отслеживания показателей веб-производительности подойдут разные инструменты, например, расширение браузера Chrome Lighthouse. Термин "основные показатели веб-производительности" (англ. *web vitals*) должен напоминать вам об основных жизненных показателях (англ. *vital signs*), таких как частота пульса и кровяное давление, поскольку они предоставляют нам информацию об исходной проблеме, которую необходимо решить.

Если вы создали свое приложение при помощи средства `create-reat-app`, то, скорее всего, оно уже содержит код, который может автоматически отслеживать его основные показатели веб-производительности. Например, файл src/index.js содержит в конце вызов функции `reportWebVitals` для создания отчета об основных показателях веб-производительности (листинг 10.31).

Листинг 10.31. Вызов функции `reportWebVitals` в конце файла src/index.js

```
import React from 'react'
import ReactDOM from 'react-dom'
import './index.css'
import App from './App'
import reportWebVitals from './reportWebVitals'
ReactDOM.render(
  <React.StrictMode>
    <App />
  </React.StrictMode>,
  document.getElementById('root')
)
reportWebVitals()
```

Функции `reportWebVitals` можно передать функцию обратного вызова, которая позволит отслеживать различные показатели в процессе исполнения приложения. Например, передадим ей `console.log`:

```
reportWebVitals(console.log)
```

В результате в консоли JavaScript браузера будут выводиться показатели веб-производительности в виде последовательности объектов JSON (рис. 10.26).

Но отслеживание веб-производительности в действительности так не осуществляется. Лучше отправлять данные обратно на сервер для хранения. Например, можно отправлять конечную точку API-интерфейса методом `POST`, как показано в листинге 10.32.

```
[HMR] Waiting for update signal from WDS...
▸{name: "TTFB", value: 8895.58000001125, delta: 8895.58000001125, entries: Array(1), id: "v1-1619375289115-2756702094238"}
▸{name: "FCP", value: 12343.090000009397, delta: 12343.090000009397, entries: Array(1), id: "v1-1619375289114-8463971860890"}
▸{name: "FID", value: 12.084999994840473, delta: 12.084999994840473, entries: Array(1), id: "v1-1619375289114-6129894643559"}
▸{name: "LCP", value: 12343.09, delta: 12343.09, entries: Array(1), id: "v1-1619375289114-8994253857417"}
```

Рис. 10.26. Основные показатели веб-производительности, выводимые в консоль JavaScript

Листинг 10.32. Отправка конечной точки API-интерфейса методом POST

```
reportWebVitals((vital) => {
  fetch('/trackVitals', {
    body: JSON.stringify(vital),
    method: 'POST',
    keepalive: true,
  })
})
```

Многие браузеры оснащены встроенной функцией записи основных показателей веб-производительности. Если пользователь уходит со страницы, браузер отменяет обычные сетевые запросы, например запросы, связанные с вызовом функции `fetch`. Если учесть, что большинство важных параметров веб-производительности связаны с загрузкой страницы, было бы неразумно потерять эти показатели. По этой причине целесообразно рассмотреть использование функции `navigator.sendBeacon`, если есть такая возможность (листинг 10.33).

Листинг 10.33. Использование функции navigator.sendBeacon

```
reportWebVitals((vital) => {
  if (navigator.sendBeacon) {
    navigator.sendBeacon('/trackVitals', JSON.stringify(vital))
  } else {
    fetch('/trackVitals', {
      body: JSON.stringify(vital),
      method: 'POST',
      keepalive: true,
    })
  }
})
```

Если пользователь откроет страницу, но затем, не дождавшись ее загрузки, уйдет с нее, функция `navigator.sendBeacon` сможет завершить отправку своего запроса `POST`.

Обсуждение

Для записи основных показателей веб-производительности на рынке предлагаются коммерческие службы отслеживания, например sentry.io (**https://sentry.io**). Если у вас установлена система мониторинга производительности, то ее можно будет настроить на использование основных показателей веб-производительности для предоставления дополнительных возможностей мониторинга производительности вашей системы.

Наконец, рассмотрите возможность мониторинга основных показателей веб-производительности с помощью средства Google Analytics, как описывается на веб-сайте средства `create-react-app` (**https://oreil.ly/wImZt**).

ГЛАВА 11

Прогрессивные веб-приложения

Прогрессивными веб-приложениями (ПВП) называются веб-приложения, которые ведут себя подобно традиционным локально устанавливаем приложениям. Они могут работать в офлайновом режиме, интегрироваться с локальной системой уведомлений, а также способны выполнять длительные фоновые процессы, которые продолжают исполняться даже после ухода посетителя с веб-сайта. Прогрессивными эти приложения называются потому, что они плавно снижают уровень своей функциональности, если какая-либо возможность не поддерживается текущим браузером.

В данной главе мы рассматриваем почти исключительно один аспект ПВА: сервис-воркеры. Иногда термин "прогрессивное веб-приложение" употребляется для описания любого браузерного приложения с большим объемом возможностей JavaScript. Но на самом деле если приложение не использует сервис-воркеров, то это не ПВП.

По сути, сервис-воркер — это локальный сервер для данного приложения. Сервер бэкенда представляет собой механизм распределения программного обеспечения и поставщика "живых" служб предоставления данных, но в действительности всем управляет сервис-воркер, поскольку он обеспечивает доступ к сети. Он принимает решения, каким образом удовлетворять сетевые запросы — обращением к бэкенд-серверу или к своему локальному кешу. При отсутствии сетевого подключения он может заменить сетевые ресурсы локальными заглушками. При работе в офлайновом режиме он даже может ставить в очередь запросы на обновление данных, а при возобновлении сетевого подключения синхронизировать их с сервером бэкенда.

Эта тема хорошо подходит для последней главы, поскольку мы испытали наибольшее удовольствие от работы над ней. Сервис-воркеры являются одной из наиболее захватывающих возможностей современных браузеров. Надеемся, что вам понравится изучение подробностей их работы.

11.1. Создаем сервис-воркеры посредством Workbox

ЗАДАЧА

Прогрессивные веб-приложения могут продолжать функционировать даже в офлайновом режиме. Они могут кешировать любой требуемый код или содержимое,

которые не подвергаются воздействию обновления страницы пользователем. Они могут исполнять фоновые операции независимо от кода, исполняемого в браузере.

Все это возможно благодаря использованию в ПВП сервис-воркеров. Сервис-воркер — это что-то наподобие веб-воркера. Веб-воркер представляет собой фрагмент кода JavaScript, который выполняется в отдельном потоке от кода JavaScript, исполняющегося в веб-странице. А сервис-воркеры — это специализированные веб-воркеры, которые могут перехватывать сетевой обмен между веб-страницей и сервером, что дает им громадный контроль над зарегистрировавшей их страницей. Сервис-воркер можно рассматривать как некий тип локального прокси-сервиса, доступного после отключения от сети.

Наиболее часто сервис-воркеры применяются для кеширования содержимого на локальном компьютере. Браузеры кешируют большую часть отображаемого ими содержимого, но сервис-воркер делает это намного более агрессивно. Например, принудительное обновление в браузере часто заставляет его загружать содержимое из сети, но не влияет на сервис-воркеры, независимо от количества таких обновлений.

На рис. 11.1 показан принцип работы сервис-воркера.

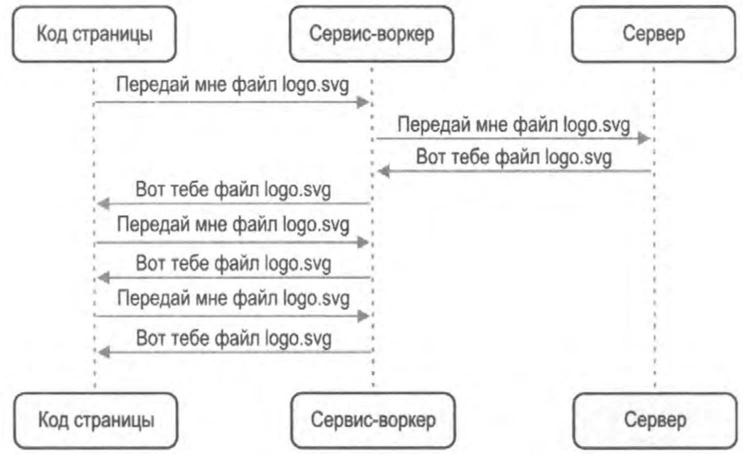


Рис. 11.1. Сервис-воркер перехватывает все сетевые запросы

В случае, показанном на рис. 11.1, сервис-воркер кеширует файлы при их первой загрузке. При последующих запросах файла logo.svg сервис-воркер будет возвращать этот файл из своего кеша, а не из сети.

Подход, применяемый сервис-воркером для кеширования данных и принятия решений о возврате их из кеша или сети, называется стратегией. В этой главе мы рассмотрим различные стандартные стратегии сервис-воркеров.

Сервис-воркеры хранятся на сервере в виде отдельных файлов JavaScript, которые браузер загружает из определенного URL-адреса и устанавливает. Нет никаких запретов на создание своего сервис-воркера и установки его в общедоступной папке

приложения, но подход с написанием сервис-воркера с чистого листа чреват несколькими проблемами.

Прежде всего, задача создания сервис-воркера кошмарно трудная. Сервис-воркеры не только содержат сложный код, но также имеют замысловатые жизненные циклы. Чрезвычайно легко ошибиться и получить сервис-воркер, который не будет загружаться или будет кешировать не те файлы. Что еще хуже, можно создать такой сервис-воркер, который полностью изолирует ваше приложение от сети.

Кроме того, сервис-воркеры можно использовать для предварительного кеширования кода приложения. Для приложений React это просто фантастическая возможность. Вместо загрузки нескольких сотен мегабайтов кода JavaScript сервис-воркер может возвратить его за долю секунды из локального кеша. Это означает, что приложение может запуститься почти моментально даже на маломощном устройстве с сетевым подключением плохого качества.

Но кеширование кода тоже не так просто осуществить. Предположим, что наше приложение React содержит следующие сгенерированные файлы с кодом JavaScript (листинг 11.1).

Листинг 11.1. Файл приложения с кодом JavaScript

```
$ ls build/static/js/
2.d106afb5.chunk.js              2.d106afb5.chunk.js.map
3.9e79b289.chunk.js.map          main.095e14c4.chunk.js.map
runtime-main.b175c5d9.js.map     2.d106afb5.chunk.js.LICENSE.txt
3.9e79b289.chunk.js              main.095e14c4.chunk.js
runtime-main.b175c5d9.js
$
```

Чтобы выполнить предварительное кеширование этих файлов, сервис-воркер должен знать их названия, поскольку он будет загружать файлы в фоновом режиме, даже до того, как браузер их запросит. Поэтому созданный вручную сервис-воркер должен будет содержать названия всех файлов, которые он будет предварительно кешировать.

Но что получится, если вы слегка измените исходный код, а затем снова выполните сборку приложения (листинг 11.2)?

Листинг 11.2. Файлы JavaScript слегка модифицированного приложения

```
$ yarn run build
$ ls build/static/js/
2.d106afb5.chunk.js              2.d106afb5.chunk.js.map
3.9e79b289.chunk.js.map           main.f5b66cc7.chunk.js.map
runtime-main.b175c5d9.js.map     2.d106afb5.chunk.js.LICENSE.txt
3.9e79b289.chunk.js              main.f5b66cc7.chunk.js
runtime-main.b175c5d9.js
$
```

Как видим, файлы JavaScript новой версии сборки будут иметь другие названия. Это означает, что вам нужно будет обновить сценарий сервис-воркера последними версиями сгенерированных названий файлов.

Так как же можно создать стабильные сервис-воркеры, которые всегда синхронизированы с последней версией кода приложения?

РЕШЕНИЕ

Поставленную задачу можно решить при помощи набора инструментов от компании Google, называющегося Workbox (**https://oreil.ly/9dPXh**). Этот набор инструментов позволяет генерировать сервис-воркеры, содержащие актуальную информацию о самых последних версиях файлов приложения.

Набор Workbox содержит коллекцию стандартных стратегий для обработки подробностей общих случаев использования сервис-воркеров. Например, выполнить предварительное кеширование приложения можно посредством всего лишь одной строки кода в Workbox.

Чтобы разобраться с работой набора Workbox, начнем с исследования примера приложения, окно которого изображено на рис. 11.2.

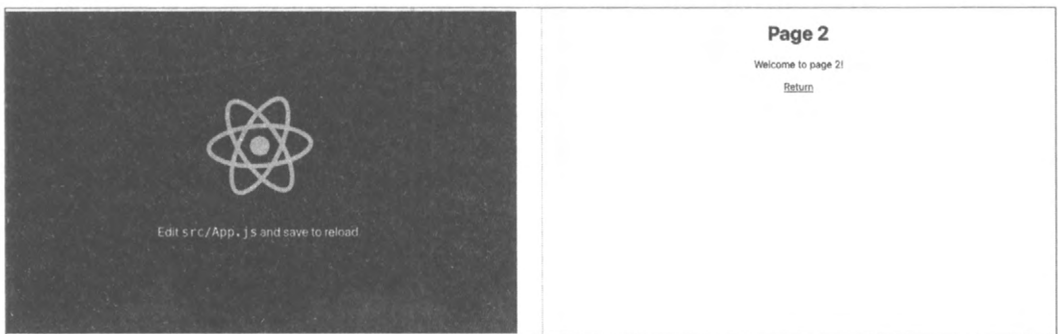


Рис. 11.2. Демонстрационное приложение содержит две страницы

Это простое двухстраничное приложение на основе стандартного приложения, сгенерированного средством `create-react-app`. Мы создадим сервис-воркер, который будет выполнять предварительное кеширование всего кода приложения и всех его файлов.

Начнем с установки нескольких библиотек из Workbox:

```
$ yarn add workbox-core
$ yarn add workbox-precaching
$ yarn add workbox-routing
```

Назначение каждой из этих библиотек будет рассмотрено в процессе создания нашего сервис-воркера.

В приложении создадим для сервис-воркера новый файл service-worker.js (листинг 11.3). Этот файл можно разместить в той же самой папке, что и остальные файлы кода приложения.

Листинг 11.3. Содержимое файла service-worker.js для сервис-воркера

```
import { clientsClaim } from 'workbox-core'
import { precacheAndRoute } from 'workbox-precaching'
clientsClaim()
precacheAndRoute(self.__WB_MANIFEST)
```

При создании сервис-воркера вручную этот файл нужно поместить в папку, где хранится другое статическое содержимое. Например, в приложении `create-react-app` это будет папка public.

Наш сервис-воркер будет выполнять предварительное кеширование всего кода приложения. Это означает, что он будет автоматически кешировать все файлы CSS, JavaScript, HTML и изображений, входящие в состав приложения.

Наш сервис-воркер вызывает функцию `clientsClaim` библиотеки `workbox-core`, которая сделает его контроллером всех клиентов в пределах его области видимости. Клиентами являются веб-страницы, а областью видимости — любая веб-страница, URL-адрес которой находится в том же пути, что и сервис-воркер. Набор Workbox создаст наш сервис-воркер с путем https://host/service-worker.js, означая, что он будет контроллером всех страниц, адрес которых начинается с https://host/.

Функция `precacheAndRoute` будет заниматься всеми непривлекательными подробностями процесса предварительного кеширования. Она создает и управляет локальным кешем, а также перехватывает сетевые запросы на файлы приложения, загружая их из локального кеша, а не из сети.

Сервис-воркеры работают только при их загрузке через протокол HTTPS. Но большинство браузеров позволяют исключение этому правилу для сайтов, загружаемых из адреса *localhost*. В целях безопасности браузеры не будут исполнять сервис-воркеры в приватных вкладках.

После создания сервис-воркера его нужно зарегистрировать из основного кода приложения. Регистрация — это сложный процесс, но, к счастью, практически почти всегда одинаковый. Создав код регистрации для одного приложения, его можно использовать без изменения для регистрации других приложений. Кроме того, если для создания приложения применяется шаблон `cra-template-pwa`, то он сгенерирует код регистрации для вас[1].

Тем не менее полезно знать подробности процесса регистрации. Это позволит вам получить представление о жизненном цикле сервис-воркеров, что, в свою очередь, намного облегчит понимание кажущегося странным поведения, возникающего после развертывания приложения.

В основной папке приложения создайте файл registerWorker.js, содержащий код, приведенный в листинге 11.4.

[1] См. *раздел 11.2*.

Листинг 11.4. Файл registerWorker.js для регистрации сервис-воркера

```
const register = (pathToWorker, onInstall, onUpdate, onError) => {
   // We will write this code shortly
}
const registerWorker = () => {
   register(
      '/service-worker.js',
      (reg) => console.info('Service worker installed', reg),
      (reg) => console.info('Service worker updated', reg),
      (err) => console.error('Service worker failed', err)
   )
}
export default registerWorker
```

Пока что оставим функцию `register` пустой.

В приложении функция `registerWorker` будет вызываться в файле index.js, как показано в листинге 11.5.

Листинг 11.5. Вызов функции `registerWorker`

```
import React from 'react'
import ReactDOM from 'react-dom'
import './index.css'
import App from './App'
import registerWorker from './registerWorker'
ReactDOM.render(
   <React.StrictMode>
      <App />
   </React.StrictMode>,
   document.getElementById('root')
)
registerWorker()
```

Функция `registerWorker` вызывает функцию `register`, передавая ей путь нашего сгенерированного сервис-воркера: service-worker.js.

Теперь можно приступать к созданию функции `register` (листинг 11.6).

Листинг 11.6. Код функции `register`

```
const register = (pathToWorker, onInstall, onUpdate, onError) => {
   if (
      process.env.NODE_ENV === 'production' &&
      'serviceWorker' in navigator
   ) {
      const publicUrl = new URL(
         process.env.PUBLIC_URL,
```

```
            window.location.href
        )
        if (publicUrl.origin !== window.location.origin) {
            return
        }
        // Сюда вставляется код для загрузки и регистрации
    }
}
```

В ней мы сначала проверяем, что приложение находится в рабочем режиме и что браузер может исполнять сервис-воркеры. Слово "прогрессивное" в названии *прогрессивное веб-приложение* означает, что прежде чем использовать возможность, мы всегда должны проверять ее наличие. Почти все браузеры (за исключением Internet Explorer) поддерживают сервис-воркеры, но при отсутствии такой поддержки браузером мы можем полностью пропустить загрузку сервис-воркера. В таком случае приложение не сможет функционировать в офлайновом режиме, но в остальном должно работать должным образом.

Код также содержит дополнительную проверку на исполнение по указанному адресу `PUBLIC_URL` приложения, чтобы избежать междоменных проблем, которые возникают при загрузке кода из сетей распределения содержимого[2].

Здесь мы создаем упрощенную версию кода из библиотеки `cra-template-pwa`. Дополнительная информация предоставляется на веб-сайте GitHub по адресу **https://oreil.ly/dKJE0**.

Теперь можно загрузить и зарегистрировать сервис-воркер, как показано в листинге 11.7.

Листинг 11.7. Загрузка и регистрация сервис-воркера

```
const register = (pathToWorker, onInstall, onUpdate, onError) => {
    if (
        process.env.NODE_ENV === 'production' &&
        'serviceWorker' in navigator
    ) {
        const publicUrl = new URL(
            process.env.PUBLIC_URL,
            window.location.href
        )
        if (publicUrl.origin !== window.location.origin) {
            return
        }
        window.addEventListener('load', async () => {
            try {
```

[2] Здесь мы создаем упрощенную версию кода из библиотеки `cra-template-pwa`. Дополнительная информация предоставляется на веб-сайте GitHub по адресу **https://oreil.ly/dKJE0**.

```
        const registration = await navigator.serviceWorker.register(
          process.env.PUBLIC_URL + pathToWorker
        )
        // Сюда вставляется код для проверки хода исполнения
      } catch (err) {
        if (onError) {
          onError(err)
        }
      }
    })
  }
}
```

Когда мы знаем, что веб-страница загрузилась, можно зарегистрировать сервис-воркер, передавая функции `navigator.serviceWorker.register` его полный URL-адрес: https://host/service-worker.js.

Функция возвращает объект `registration`, который можно использовать для отслеживания и управления сервис-воркером. Например, при помощи объекта регистрации можно узнать об обновлении или установке сервис-воркера, как показано в листинге 11.8.

Листинг 11.8. Использование объекта `registration` для получения информации об установке или обновлении сервис-воркера

```
const register = (pathToWorker, onInstall, onUpdate, onError) => {
  if (
    process.env.NODE_ENV === 'production' &&
    'serviceWorker' in navigator
  ) {
    const publicUrl = new URL(
      process.env.PUBLIC_URL,
      window.location.href
    )
    if (publicUrl.origin !== window.location.origin) {
      return
    }
    window.addEventListener('load', async () => {
      try {
        const registration = await navigator.serviceWorker.register(
          process.env.PUBLIC_URL + pathToWorker
        )
        registration.onupdatefound = () => {
          const worker = registration.installing
          if (worker) {
            worker.onstatechange = () => {
              if (worker.state === 'installed') {
```

```
                        if (navigator.serviceWorker.controller) {
                            if (onUpdate) {
                                onUpdate(registration)
                            }
                        } else {
                            if (onInstall) {
                                onInstall(registration)
                            }
                        }
                    }
                }
            }
        }
    } catch (err) {
        if (onError) {
            onError(err)
        }
    }
  })
 }
}
```

Обработчик `onupdatefound` исполняется, когда браузер начинает устанавливать сервис-воркер. После того как браузер установит сервис-воркер, можно проверить, продолжает ли исполняться предыдущий сервис-воркер, вызвав для этого функцию `navigator.serviceWorker.controller`. Если нет, значит, это установка нового сервис-воркера, а не обновление старого.

Способ обновления сервис-воркеров является, пожалуй, одним из их наиболее трудных для понимания аспектов. Если страница уже находится под управлением сервис-воркера, то браузер поставит нового сервис-воркера в *состояние ожидания*. Это означает, что новый сервис-воркер *не будет делать абсолютно ничего* до тех пор, пока не прекратит работу старый сервис-воркер. Сервис-воркер прекращает работу тогда, когда пользователь закрывает все управляемые этим воркером страницы. В результате после обновления сервис-воркера новый код не будет исполняться, пока вы не закроете и снова не откроете страницу.

Этот процесс может сбивать с толку любого разработчика, выполняющего ручное тестирование какой-либо возможности сервис-воркера.

Прежде чем приступать к созданию приложения, нам нужно выполнить настройку инструментальных средств для преобразования нашего файла исходного кода service-worker.js в плотно упакованный код сценария сервис-воркера.

Для сборщика пакетов Webpack нужно установить подключаемый модуль `workbox-webpack-plugin`:

```
$ yarn install -D workbox-webpack-plugin
```

Для приложений, созданных при помощи средства `create-react-app`, устанавливать и конфигурировать модуль Workbox Webpack Plugin не нужно, поскольку это средство автоматически устанавливает и конфигурирует данный модуль.

Теперь добавим в файл конфигурации webpack.config.js код из листинга 11.9.

Листинг 11.9. Код для добавления в файл конфигурации webpack.config.js

```
const { InjectManifest } = require('workbox-webpack-plugin')
module.exports = {
  ....
  plugins: [
    ....
    new InjectManifest({
      swSrc: './src/service-worker.js',
    }),
  ],
}
```

Согласно указаниям данной конфигурации сборщик пакетов Webpack создаст сервис-воркера из файла src/service-worker.js. Также он создаст в собранном приложении файл asset-manifest.json, содержащий список всех файлов приложения. Информация в этом файле будет использоваться сервис-воркером при предварительном кешировании приложения.

Теперь можно создать само приложение, исполнив следующую команду:

```
$ yarn run build
```

В результате в папке build будет создан файл сервис-воркера service-worker.js, а также файл asset-manifest.json (листинг 11.10).

Листинг 11.10. Содержимое папки build после создания приложения

```
asset-manifest.json logo192.png service-worker.js.map
favicon.ico manifest.json static
index.html robots.txt
logo512.png service-worker.js
```

Файл asset-manifest.json будет содержать код, наподобие показанного в листинге 11.11.

Листинг 11.11. Содержимое файла asset-manifest.json

```
{
  "files": {
  "main.css": "/static/css/main.8c8b27cf.chunk.css",
  "main.js": "/static/js/main.f5b66cc7.chunk.js",
```

```
  "main.js.map": "/static/js/main.f5b66cc7.chunk.js.map",
  "runtime-main.js": "/static/js/runtime-main.b175c5d9.js",
  "runtime-main.js.map": "/static/js/runtime-main.b175c5d9.js.map",
  "static/js/2.d106afb5.chunk.js": "/static/js/2.d106afb5.chunk.js",
  "static/js/2.d106afb5.chunk.js.map": "/static/js/2.d106afb5.chunk.js.map",
  "static/js/3.9e79b289.chunk.js": "/static/js/3.9e79b289.chunk.js",
  "static/js/3.9e79b289.chunk.js.map": "/static/js/3.9e79b289.chunk.js.map",
  "index.html": "/index.html",
  "service-worker.js": "/service-worker.js",
  "service-worker.js.map": "/service-worker.js.map",
  "static/css/main.8c8b27cf.chunk.css.map":
    "/static/css/main.8c8b27cf.chunk.css.map",
  "static/js/2.d106afb5.chunk.js.LICENSE.txt":
    "/static/js/2.d106afb5.chunk.js.LICENSE.txt",
  "static/media/logo.6ce24c58.svg": "/static/media/logo.6ce24c58.svg"
  },
  "entrypoints": [
    "static/js/runtime-main.b175c5d9.js",
    "static/js/2.d106afb5.chunk.js",
    "static/css/main.8c8b27cf.chunk.css",
    "static/js/main.f5b66cc7.chunk.js"
  ]
}
```

Теперь можно исполнять приложение. Но мы не можем просто запустить на исполнение сервер для разработки как обычно:

```
$ yarn run start
```

Такой запуск приложения возможен только в режиме разработки, и сервис-воркеры не запустятся. Но нам нужно запустить сервер на содержимом папки build. Легче всего сделать это, установив пакет `serve` глобально, а затем исполняя его по папке build, как показано в листинге 11.12.

Листинг 11.12. Установка и запуск пакета `serve`

```
$ npm install -s serve
$ serve -s build/

   ┌────────────────────────────────────────────────┐
   │                                                │
   │  Serving!                                      │
   │                                                │
   │  - Local: http://localhost:5000                │
   │  - On Your Network: http://192.168.1.14:5000   │
   │                                                │
   │  Copied local address to clipboard!            │
   │                                                │
   └────────────────────────────────────────────────┘
```

Опция `-s` предназначена для исполнения одностраничных приложений. Если сервер не сможет найти требуемый файл, то он возвратит файл build/index.html.

Теперь запустим браузер и откроем страницу по адресу **http://localhost:5000**. Запустив таким образом приложение, откройте окно инструментов разработчика, а в нем вкладку **Application**. Здесь в разделе **Service Workers** должно отображаться исполнение сценария service-worker.js (рис. 11.3).

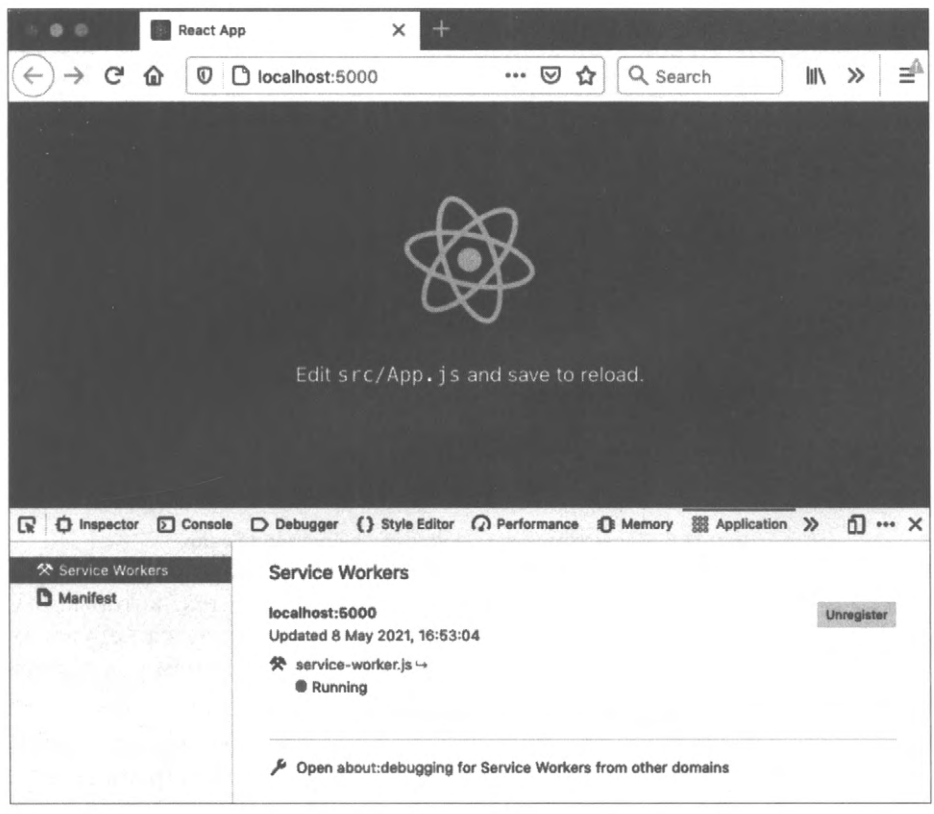


Рис. 11.3. Приложение с установленным и исполняющимся сервис-воркером

Сервис-воркер загружает все файлы приложения в локальный кеш, в результате чего при следующей загрузке страницы данные файлы будут взяты из локального кеша, а не по сети с сервера. Это можно наблюдать, если в окне инструментов разработчика переключиться на вкладку **Network**, а затем перезагрузить страницу (рис. 11.4).

Сервис-воркер предоставит все сетевые ответы, за исключением тех, которые выходят за рамки его области видимости. Любой файл уровня сайта, а не страницы, например значки *favicon*, будет и дальше загружаться обычным способом с сервера.

Итак, сервис-воркер возвращает файлы из локального кеша. В браузере Chrome содержимое локального кеша можно просмотреть на вкладке **Application**, а в Firefox — на вкладке **Storage** (рис. 11.5).

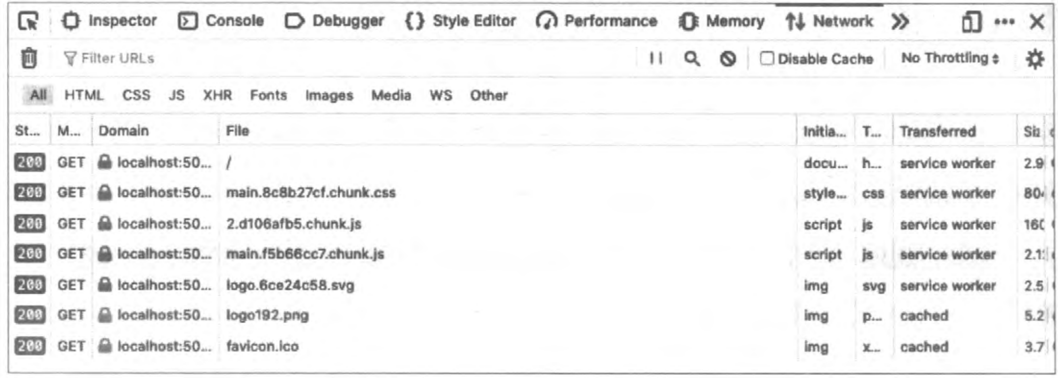


Рис. 11.4. После обновления страницы все файлы предоставляются сервис-воркером

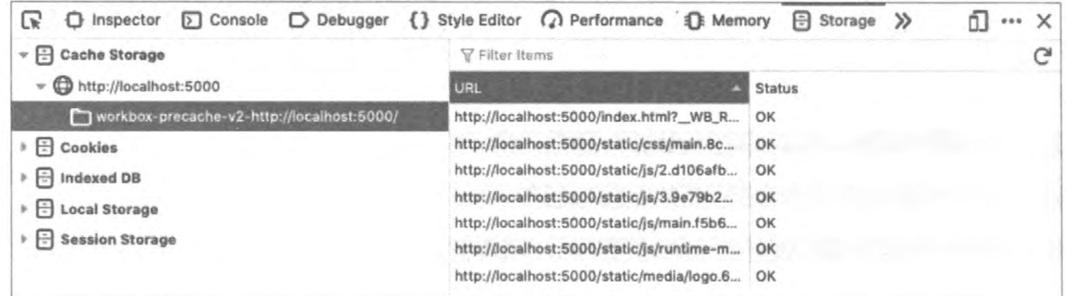


Рис. 11.5. Содержимое локального кеша браузера Firefox

Кеш содержит копии не всех файлов приложения, а только тех, которые были запрошены приложением. Таким образом предотвращается загрузка ненужных файлов, и файлы загружаются в кеш только тогда, когда браузер или код приложения запросит их.

Поэтому при первоначальной загрузке приложения кеш может быть пустым. Загрузка зависит от того, когда активируется сервис-воркер. Если страница загружена до активирования сервис-воркера, то он не будет перехватывать сетевые запросы и кешировать ответы на них. В результате, чтобы началось кеширование, нужно будет перезагрузить страницу.

Чтобы убедиться в том, что файлы действительно поставляются сервис-воркером, можно остановить сервер и обновить веб-страницу. Вы должны увидеть, что даже при отсутствующем сервере страница загружается как обычно (рис. 11.6).

Теперь приложение React следует рассматривать как локальное, а не сетевое приложение, поскольку оно обслуживается сервис-воркером, а не сервером бэкенда. Оно даже позволит выполнить переход на вторую страницу (рис. 11.7).

Использование в приложении разделения кода может отрицательно сказаться на офлайновой функциональности. Если код для отображения второй страницы хранился в отдельном файле JavaScript, который не был загружен при первоначальном запуске приложения, браузер не сможет возвратить его из локального кеша. Он станет доступным лишь после посещения браузером страницы в режиме онлайн, когда возобновится доступ к серверу.

Рис. 11.6. Даже при отсутствии сервера страница обновляется без проблем

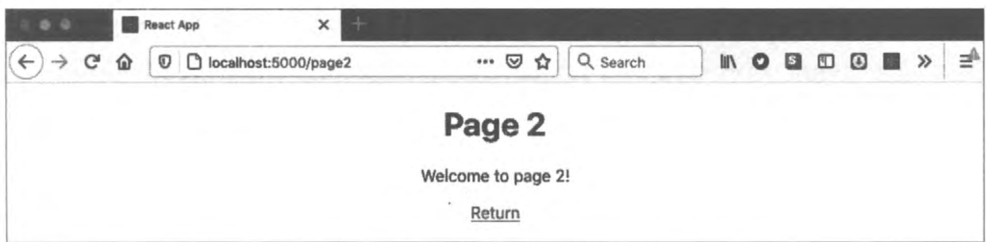


Рис. 11.7. При отсутствии сервера можно даже переходить с одной страницы на другую

Пока мы находимся на второй странице, мы можем разобраться с текущей проблемой с сервис-воркером. Убедитесь в том, что сервер *не работает*, и перейдите на вторую страницу. Страница должна загрузиться нормально. Далее перезагрузите страницу. Теперь вместо второй страницы браузер должен отобразить страницу ошибки (рис. 11.8).

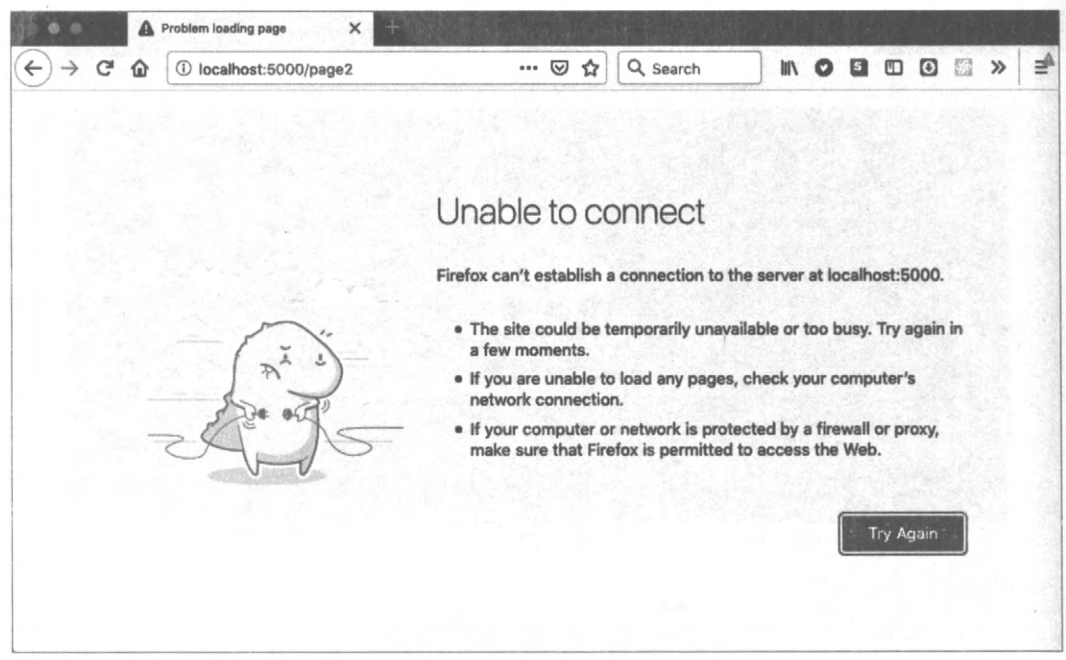


Рис. 11.8. При отсутствии сервера вторая страница не будет перезагружаться

Мы можем перезагружать домашнюю страницу в офлайновом режиме, так в чем же проблема с перезагрузкой второй страницы? Причина в том, что мы имеем дело с одностраничным приложением. При переходе на вторую страницу браузер не загружает новую веб-страницу с сервера. Вместо этого он использует интерфейс History API, чтобы обновить URL в строке адреса, а затем модифицировать модель DOM, чтобы отобразить вторую страницу.

Но при перезагрузке страницы браузер отправляет серверу новый запрос ресурса по адресу **http://localhost:5000/page2**. Когда сервер доступен, он возвращает содержимое файла index.html для всех запросов страниц, а далее уже маршрутизатор React отрисовывает компоненты, чтобы создать представление второй страницы.

Но при недоступном сервере этот процесс разваливается. Сервис-воркер не сможет ответить на запрос ресурса **http://localhost:5000/page2**, используя кешированные данные, поскольку кеш не содержит ничего для второй страницы. Поэтому сервис-воркер перенаправляет этот запрос серверу, который, как было сказано, в настоящее время недоступен. В результате возвращается страница с сообщением об ошибке.

Эту проблему можно исправить, добавив немного дополнительного кода в файл service-worker.js, как показано в листинге 11.13[3].

---

[3] Данный код основан на коде примера сервис-воркера в библиотеке `cra-template-pwa`, который мы рассмотрим в следующем разделе.

Листинг 11.13. Файл *service-worker.js* с дополнительным кодом

```
import { clientsClaim } from 'workbox-core'
import {
   createHandlerBoundToURL,
   precacheAndRoute,
} from 'workbox-precaching'
import { registerRoute } from 'workbox-routing'
clientsClaim()
precacheAndRoute(self.__WB_MANIFEST)
const fileExtensionRegexp = new RegExp('/[^/?]+\.[^/]+$')
registerRoute(({ request, url }) => {
   if (request.mode !== 'navigate') {
      return false
   }
   if (url.pathname.startsWith('/_')) {
      return false
   }
   if (url.pathname.match(fileExtensionRegexp)) {
      return false
   }
   return true
}, createHandlerBoundToURL(process.env.PUBLIC_URL + '/index.html'))
```

Теперь мы регистрируем явный маршрут, задействовав модуль `workbox-routing`. Маршрут определяет, как сервис-воркер будет обрабатывать запросы для набора путей. Мы регистрируем новый маршрут, используя функцию фильтрации и обработчик из кода предыдущего примера. Функция фильтрации передается в первом значении функции `registerRoute`. Если этот маршрут относится к данному запросу, то она возвратит значение `true`. Функция фильтрации в листинге 11.13 обрабатывает все запросы перехода на новую веб-страницу. Таким образом, если открыть в браузере страницу по адресу **http://localhost:5000/** или **http://localhost:5000/page2**, то этот маршрут возвратит одну и ту же копию файла index.html.

Функция `createHanderBoundToURL` создаст обработчик для обработки любого из этих запросов так, будто они были запросами ресурса по адресу **http://localhost: 5000/index.html**. Если перезагрузить приложение, когда открыта вторая страница, то сервис-воркер должен загрузить содержимое HTML таким же образом, как он это делает, когда открыта домашняя страница.

Давайте попробуем проделать это. Сохраним изменения в файле service-worker.js и снова выполним сборку приложения:

```
$ yarn run build
```

Далее обеспечим наличие сервера:

```
$ serve -s build/
```

Если теперь открыть в браузере страницу по адресу **http://localhost:5000**, то должна отобразиться домашняя страница приложения. Открыв окно инструментов разработчика, мы должны увидеть, что была загружена новая версия сервис-воркера, но при этом его старая версия продолжает исполняться (рис. 11.9).

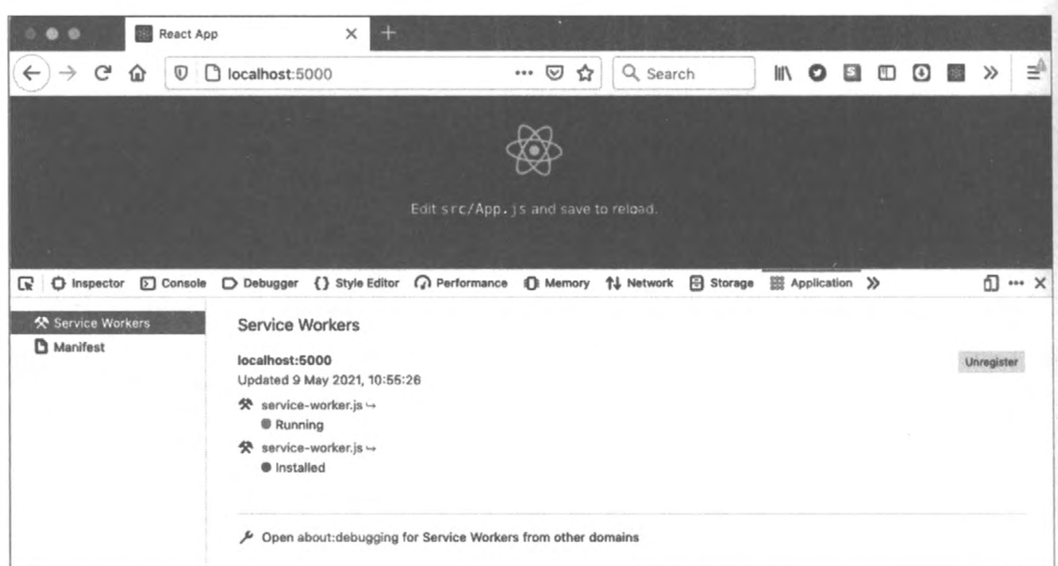


Рис. 11.9. В приложении присутствует как старая, так и новая версия сервис-воркера

Предыдущая версия сервис-воркера продолжает управлять приложением. Браузер установил новую версию воркера, но она находится в состоянии ожидания. Она не получит управления до тех пор, пока не будет удален старый сервис-воркер. Чтобы это случилось, нужно закрыть приложение, а затем снова открыть его (рис. 11.10).

Рис. 11.10. Новый сервис-воркер активируется перезапуском приложения

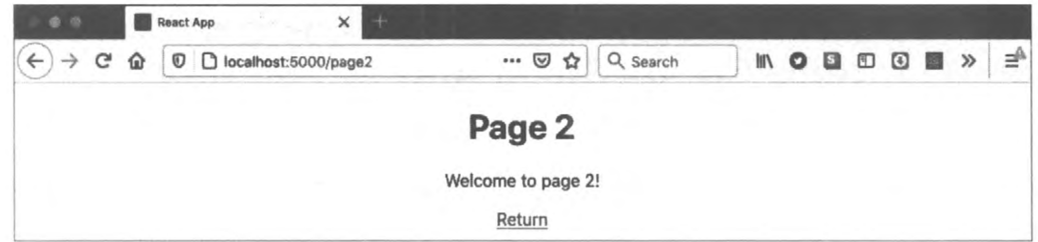


Рис. 11.11. После регистрации обработчика маршрута вторая страница без проблем перезагружается в офлайновом режиме

Если теперь остановить локальный сервер и перейти на вторую страницу, то она должна перезагружаться без каких бы то ни было проблем (рис. 11.11).

Обсуждение

В этом разделе мы рассмотрели довольно глубоко процесс создания, регистрации и использования сервис-воркеров. В следующем рецепте мы рассмотрим, как можно автоматически генерировать большую часть этого кода на этапе создания приложения. Но тем не менее полезно разобраться со всеми неприглядными подробностями работы сервис-воркеров. Это поможет вам понять жизненный цикл сервис-воркера: как происходит его установка браузером и как он активируется.

Мы открыли для себя, что сервис-воркеры могут сбить с толку любого, кто тестирует код вручную. Если браузер продолжает исполнять старую версию сервис-воркера, то, возможно, он будет продолжать исполнять старую версию приложения. Эта неразбериха потенциально повлечет за собой проблему тестирования, поскольку прежняя версия приложения может содержать какую-либо старую ошибку. Но поняв процесс загрузки новых сервис-воркеров и удаления старых, вы сможете быстро диагностировать проблему.

Устарелые сервис-воркеры не представляют проблем для автоматизированных браузерных тестов, которые обычно начинают исполняться с чистого состояния, без кешированного содержимого или исполняющихся сервис-воркеров.

Прогрессивные веб-приложения с сервис-воркерами представляют собой своеобразный гибрид локального и удаленного приложений. Сервер становится сервером распределения для локально установленного приложения. При обновлении приложения оно устанавливает в браузере свою новую версию, которая обычно не будет доступной, пока браузер не откроет ее.

Теперь, когда мы подробно исследовали сервис-воркеры, можно перейти к рассмотрению, как их можно быстро добавить в новое приложение.

Исходный код данного рецепта можно загрузить на веб-сайте GitHub по адресу **https://oreil.ly/su224**.

11.2. Создание прогрессивных веб-приложений посредством Create React App

ЗАДАЧА

Для работы сервис-воркеров в приложении необходимо выполнить два условия. Первое, нужно иметь в наличии сам сервис-воркер. В *разделе 11.1* мы рассмотрели, как создавать и управлять сервис-воркерами при помощи библиотеки Workbox. Второе, требуется код для регистрации сервис-воркера в приложении. Хотя создание такого кода сопряжено со сложностями, однажды созданный регистрационный код можно использовать повторно для регистрации сервис-воркеров в других приложениях, внося в него небольшие модификации.

Но по мере развития особенностей применения сервис-воркеров было бы неплохо избавиться от необходимости создавать регистрационный код для них самостоятельно. Как мы можем это сделать?

РЕШЕНИЕ

Поставленную задачу можно решить с помощью шаблонов средства `create-react-app` для создания приложений, содержащих сервис-воркеры.

Даже если вы не планируете использовать средство `create-react-app`, лучше сначала создать приложение с его помощью, а затем добавить его код сервис-воркера в свое приложение.

Мы вкратце рассмотрели тему шаблонов приложений в *главе 1*, где создавали приложения с использованием TypeScript при помощи средства `create-react-app`. Шаблоны — это стандартный код, присоединяемый средством `create-react-app` при создании нового приложения.

Создать прогрессивное веб-приложение можно, выполнив следующую команду:

```
$ npx create-react-app appname --template cra-template-pwa
```

Чтобы создать приложение с использованием TypeScript, замените `cra-template-pwa` на `cra-template-pwa-typescript`.

В результате исполнения этой команды будет сгенерировано приложение React в папке appname. Это будет практически точно такое же приложение, как и любое другое приложение `create-react-app`, но с установленными в нем библиотеками Workbox, а также двумя дополнительными файлами исходного кода. В папке src будет находиться файл сервис-воркера service-worker.js, содержимое которого приведено в листинге 11.14.

Листинг 11.14. Содержимое файла сервис-воркера *service-worker.js*

```
import { clientsClaim } from 'workbox-core'
import { ExpirationPlugin } from 'workbox-expiration'
```

```
import {
  precacheAndRoute,
  createHandlerBoundToURL,
} from 'workbox-precaching'
import { registerRoute } from 'workbox-routing'
import { StaleWhileRevalidate } from 'workbox-strategies'
clientsClaim()
precacheAndRoute(self.__WB_MANIFEST)
const fileExtensionRegexp = new RegExp('/[^/?]+\.[^/]+$')
registerRoute(({ request, url }) => {
  if (request.mode !== 'navigate') {
    return false
  }
  if (url.pathname.startsWith('/_')) {
    return false
  }
  if (url.pathname.match(fileExtensionRegexp)) {
    return false
  }
  return true
}, createHandlerBoundToURL(process.env.PUBLIC_URL + '/index.html'))
registerRoute(
  ({ url }) =>
    url.origin === self.location.origin &&
    url.pathname.endsWith('.png'),
  new StaleWhileRevalidate({
    cacheName: 'images',
    plugins: [new ExpirationPlugin({ maxEntries: 50 })],
  })
)
self.addEventListener('message', (event) => {
  if (event.data && event.data.type === 'SKIP_WAITING') {
    self.skipWaiting()
  }
})
```

Этот сервис-воркер очень похож на сервис-воркер, который мы создали в *разделе 11.1*.

Папка src также содержит еще один новый файл — serviceWorkerRegistration.js. Поскольку его размер очень велик, мы не приводим его содержимое здесь. Назначение этого файла такое же, как и назначение файла сценария registerWorker.js, который мы создали в *разделе 11.1*, — он регистрирует наш сервис-воркер в качестве контроллера для приложения. Файл serviceWorkerRegistration.js будет очень полезен и для приложений, созданных без применения средства `create-react-app`. Он содержит несколько дополнительных возможностей, которые отсутствовали в файле регистрации, созданном нами в *разделе 11.1*. Например, предположим, что наше прило-

жение исполняется на *localhost*. В таком случае оно аннулирует регистрацию всех сервис-воркеров, которые выглядят как принадлежащие другим приложениям. Это полезная возможность, если вы одновременно работаете над несколькими приложениями React.

Хотя в приложении автоматически создаются сервис-воркер и код для его регистрации, они не сконфигурированы для работы. Более того, в файле index.js приложение на самом деле отменяет регистрацию всех сервис-воркеров, как показано в листинге 11.15.

Листинг 11.15. Отмена регистрации сервис-воркеров в файле index.js

```
import React from 'react'
import ReactDOM from 'react-dom'
import './index.css'
import App from './App'
import * as serviceWorkerRegistration from './serviceWorkerRegistration'
import reportWebVitals from './reportWebVitals'
ReactDOM.render(
   <React.StrictMode>
      <App />
   </React.StrictMode>,
   document.getElementById('root')
)
serviceWorkerRegistration.unregister()
reportWebVitals()
```

Поэтому, чтобы разрешить сценарий service-worker.js, функцию `WorkerRegistration.unregister` нужно заменить функцией `serviceWorkerRegistration.register`.

Функция `serviceWorkerRegistration.register` позволяет передавать в регистрационный процесс функции обратного вызова, чтобы можно было отслеживать текущее состояние установки сервис-воркера. В частности, передается объект с функциями `onInstal` и `onUpdate`, как показано в листинге 11.15.

Листинг 11.15. Код для отслеживания текущего состояния сервис-воркера

```
serviceWorkerRegistration.register({
   onInstall: (registration) => {
      console.log('Service worker installed')
   },
   onUpdate: (registration) => {
      console.log('Service worker updated')
   },
})
```

Функции обратного вызова полезны в тех случаях, когда нужно отложить исполнение какого-либо кода до тех пор, пока браузер не установит сервис-воркер, или

если необходимо исполнить код, когда новый сервис-воркер является обновлением предыдущего. Вызов функции `onUpdate` означает, что новый сервис-воркер ожидает, пока будет удален предыдущий воркер.

Обсуждение

Материал *раздела 11.1* поможет вам понять принцип работы сервис-воркеров. При разработке настоящего приложения код, созданный по шаблонам, будет намного более детализирован и насыщен возможностями.

Исходный код данного рецепта можно загрузить на веб-сайте GitHub по адресу **https://oreil.ly/hHAC9**.

11.3. Кеширование сторонних ресурсов

ЗАДАЧА

Многие используемые в современных приложениях ресурсы предоставляются сторонними серверами, например платежные библиотеки, шрифты, изображения и т. п. Сторонние ресурсы могут потреблять большой объем пропускной способности сети, и со временем их размер может увеличиваться. Если они предоставляются медленными серверами, то ваше приложение замедлится и у вас не будет никакого контроля над этим[4].

Можно ли использовать сервис-воркеры для кеширования сторонних ресурсов?

РЕШЕНИЕ

Сервис-воркеры имеют ограниченную область видимости, поскольку им разрешается управлять только страницами, имеющими такой же путь URL. Этим объясняется тот факт, что сервис-воркеры обычно размещаются в корневой папке приложения, что позволяет им управлять всеми его страницами.

Но подобного ограничения не существует на URL-адреса, с которыми они могут устанавливать контакт. Они могут общаться с любой конечной точкой, с которой может общаться страница или код приложения. Это означает, что мы можем кешировать ресурсы, поступающие со сторонних серверов.

На рис. 11.12 показано окно приложения, использующего шрифт, загруженный с веб-сайта Google Fonts.

Этот шрифт был добавлен в приложение при помощи следующих двух строк кода в заголовке страницы:

---

[4] Недавно мы работали над проектом, полагавшимся на стороннюю платежную библиотеку. При тестировании производительности приложения было обнаружено, что эта библиотека была намного медленнее, чем самый медленный компонент приложения, не просто из-за своего большого размера, но потому, что ее серверу часто требовалось несколько сотен миллисекунд, чтобы начать загружать код.

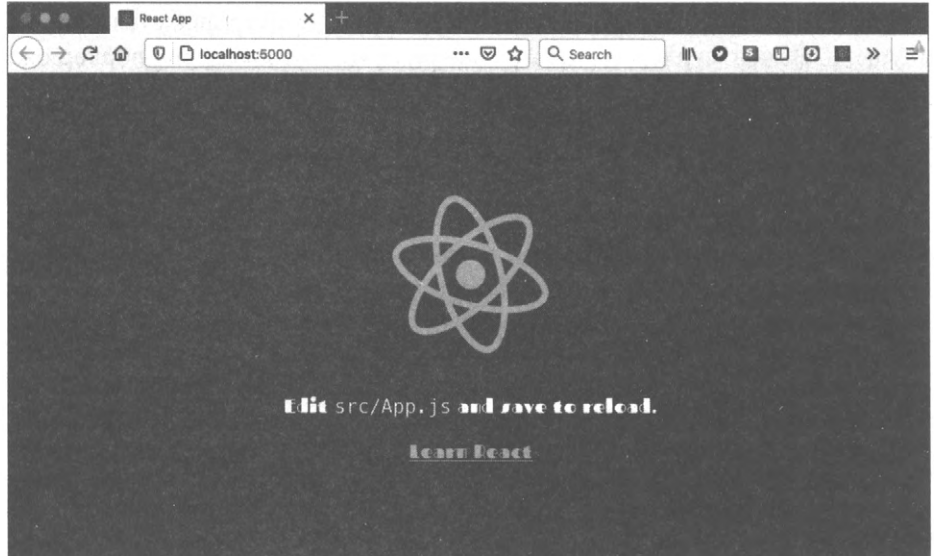


Рис. 11.12. Это приложение использует шрифт Google

```
<link rel="preconnect" href="https://fonts.gstatic.com">
<link href="https://fonts.googleapis.com/css2?family=Fascinate&display=swap"
   rel="stylesheet">
```

Первая ссылка импортирует шрифт веб-страницы, а вторая — связанную таблицу стилей.

Чтобы этот шрифт можно было кешировать в приложении, сначала нужно зарегистрировать сервис-воркер. Поскольку наше демонстрационное приложение было создано на основе шаблона библиотеки `cra-template-pwa`, то для этого нужно лишь вызвать функцию `register` в файле index.js, как показано в листинге 11.16.

Листинг 11.16. Вызов функции `register` для регистрации сервис-воркера

```
import React from 'react'
import ReactDOM from 'react-dom'
import './index.css'
import App from './App'
import * as serviceWorkerRegistration from './serviceWorkerRegistration'
import reportWebVitals from './reportWebVitals'
ReactDOM.render(
   <React.StrictMode>
      <App />
   </React.StrictMode>,
   document.getElementById('root')
)
serviceWorkerRegistration.register()
reportWebVitals()
```

Далее мы добавим в сценарий service-worker.js, содержащий сервис-воркер приложения, несколько маршрутов. Сервис-воркер использует библиотеку Workbox.

Нам нужно кешировать таблицу стилей и загружаемый шрифт.

Из *раздела 11.1* мы узнали, что можно выполнять предварительное кеширование кода приложения. Это настолько распространенная задача, что библиотека Workbox позволяет делать это одной строкой кода:

```
precacheAndRoute(self.__WB_MANIFEST)
```

Эта команда создает маршрут, по которому будет локально кешироваться код приложения. Но для кеширования сторонних ресурсов нужно еще немного поработать. Сначала создадим маршрут для кеширования таблицы стилей (листинг 11.17).

Листинг 11.17. Создание маршрута для кеширования таблицы стилей

```
registerRoute(
  ({ url }) => url.origin === 'https://fonts.googleapis.com'
  // TODO Add handler
)
```

При вызове функции `registerRoute` ей нужно передать фильтрующую функцию и обработчик. Фильтрующая функция принимает объект запроса и возвращает значение `true`, если обработчик должен его обработать. Обработчик представляет собой функцию, которая решает, каким образом выполнить запрос. Он может взять требуемые данные из кеша, передать запрос в сеть или использовать какую-либо комбинацию этих методов.

Обработчики представляют собой довольно сложные функции, но они обычно действуют согласно определенной стандартной стратегии, например проверяют наличие требуемых данных в кеше, прежде чем загружать их из сети. Библиотека Workbox содержит функции, реализующие несколько стратегий.

При загрузке таблицы стилей мы применяем стратегию `stale-while-revalidate` (**https://oreil.ly/Ct1K3**). Согласно этой стратегии, когда браузер хочет загрузить таблицу стилей Google, мы отправляем запрос для данной таблицы стилей, а также проверяем наличие копии ее файла в локальном кеше. При отсутствии таблицы стилей в кеше мы ожидаем возвращения ответа на ее запрос из Сети. Эта стратегия полезна в случае частых запросов ресурса, при этом не придавая важности наличию его самой последней версии. Наличие кешируемой версии таблицы стилей будет предпочтительнее, поскольку это будет быстрее. Но мы также всегда запрашиваем новую версию таблицы стилей из Сети. Возвращенный ответ от Google сохраняется в кеше, поэтому, даже если в этот раз мы не получим самую последнюю версию таблицы стилей, мы ее получим при ее следующей загрузке.

Код обработчика для стратегии `stale-while-revalidate` приведен в листинге 11.18.

Листинг 11.18. Реализация обработчика стратегии `stale-while-revalidate`

```
registerRoute(
  ({ url }) => url.origin === 'https://fonts.googleapis.com',
  new StaleWhileRevalidate({
    cacheName: 'stylesheets',
  })
)
```

Функция `StaleWhileRevalidate` возвращает функцию обработчика, которая сохраняет таблицу стилей в кеше `stylesheets`.

При запросе сторонних ресурсов может случиться, что запрос завершается неуспешно с сообщением об ошибке CORS (cross-origin resource sharing — совместное использование ресурсов между разными источниками). Подобная ошибка может возникать даже при возвращении стороннего ресурса с действительным заголовком CORS, поскольку запрос `GET` исходит из кода JavaScript, а не из кода HTML-страницы. Эту ошибку можно исправить, присвоив значение `anonymous` атрибуту `crossorigin` элемента HTML, использующего ресурс, например ссылке `link` для загрузки таблицы стилей.

С помощью этой же стратегии можно было бы загружать и шрифт Google. Но файлы шрифтов могут быть большого размера, а стратегия `stale-while-revalidate` всегда загружает самую последнюю версию ресурса, даже если необходимо только обновить локальный кеш.

Поэтому вместо нее мы задействуем стратегию `cache-first` (**https://oreil.ly/c8aa5**). При этом подходе мы сначала проверяем наличие ресурса в кеше и при положительном результате берем версию ресурса из кеша. В противном же случае отправляется запрос на получение ресурса из Сети. Такая стратегия предпочтительна для ресурсов большого размера. Но она имеет свою отрицательную сторону: новая версия ресурса загружается только при его отсутствии в кеше. Это означает, что новая версия может никогда не загрузиться.

По этой причине стратегия `cache-first` обычно настраивается на запрос ресурсов из кеша только в течение определенного времени. Если обработчик находит ресурс в кеше, но он слишком старый, то обработчик запрашивает ресурс из сети и кеширует полученную новую версию.

Сохраненный в кеше ресурс будет использоваться до тех пор, пока не истечет его время тайм-аута. Таким образом, в случае каких-либо временных проблем со сторонним сервером и получения ошибки 500[5], мы не хотим кешировать этот результат. Поэтому, прежде чем сохранять ответ в кеше, нам также нужно проверить статус этого ответа.

В листинге 11.19 приведен код для регистрации маршрута для кеширования шрифта Google.

[5] Internal Server Error — внутренняя ошибка сервера

Листинг 11.19. Регистрация маршрута для кеширования шрифта Google

```
registerRoute(
  ({ url }) => url.origin === 'https://fonts.gstatic.com',
  new CacheFirst({
    cacheName: 'fonts',
    plugins: [
      new CacheableResponsePlugin({
        statuses: [0, 200],
      }),
      new ExpirationPlugin({
        maxAgeSeconds: 60 * 60 * 24 * 7,
        maxEntries: 5,
      }),
    ],
  })
)
```

Этот код сохраняет вплоть до пяти файлов шрифтов в локальном кеше `fonts`. Кешированные копии обладают тайм-аутом длительностью в одну неделю, а ответ кешируется только если он имеет статус `200` или `0`. Статус `0` означает проблему CORS с запросом, и в этом случае мы кешируем ответ. Ошибку CORS нельзя устранить, не выполнив соответствующие модификации кода, поэтому, сохраняя ответ в кеше, мы избегаем отправки дальнейших запросов, которые обречены на неуспех.

Обсуждение

Кеширование сторонних ресурсов может существенно улучшить производительность приложения, но, что намного более важно, сделает ресурсы доступными в офлайновом режиме. Если приложение не может получить какой-либо косметический ресурс типа файла шрифта, то это не слишком важно. Но если сторонний ресурс необходим для генерирования платежной формы, то было бы целесообразным продолжить работу, даже если сетевое подключение временно отсутствует.

Исходный код данного рецепта можно загрузить на веб-сайте GitHub по адресу **https://oreil.ly/QaFYG**.

11.4. Автоматическая перезагрузка воркеров

ЗАДАЧА

Метод обновления сервис-воркеров может сбивать с толку любого, кто применяет или тестирует использующее их приложение. Приложение загружает новую версию обновленного сервис-воркера и присваивает ему статус **Installed**, как показано на рис. 11.13.

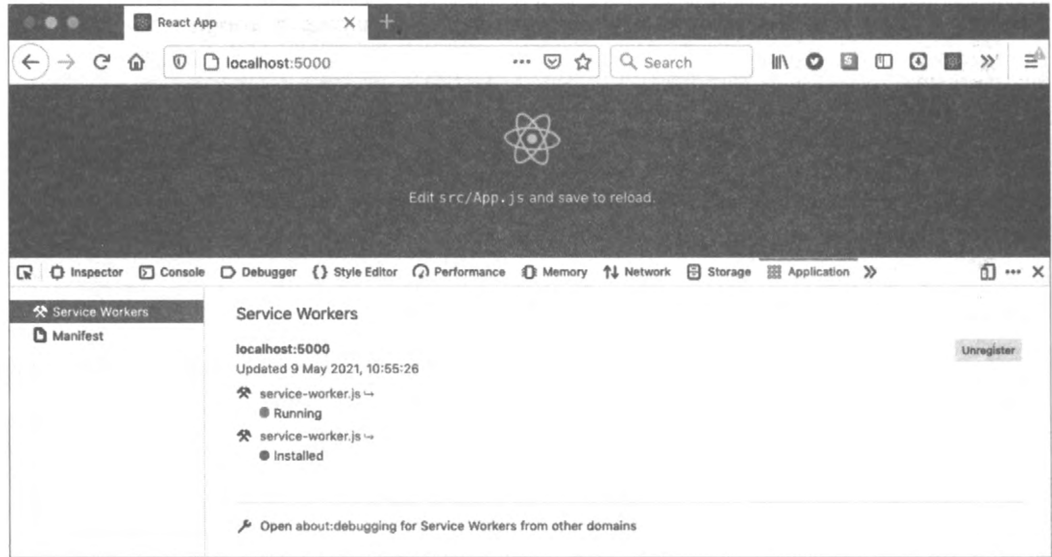


Рис. 11.13. После установки обновленного сервис-воркера его старая версия продолжает исполняться

Старая версия сервис-воркера будет удалена только после того, как пользователь закроет приложение, а затем снова откроет его. В результате старый сервис-воркер удаляется, а новый выходит из режима ожидания и начинает исполняться (рис. 11.14).

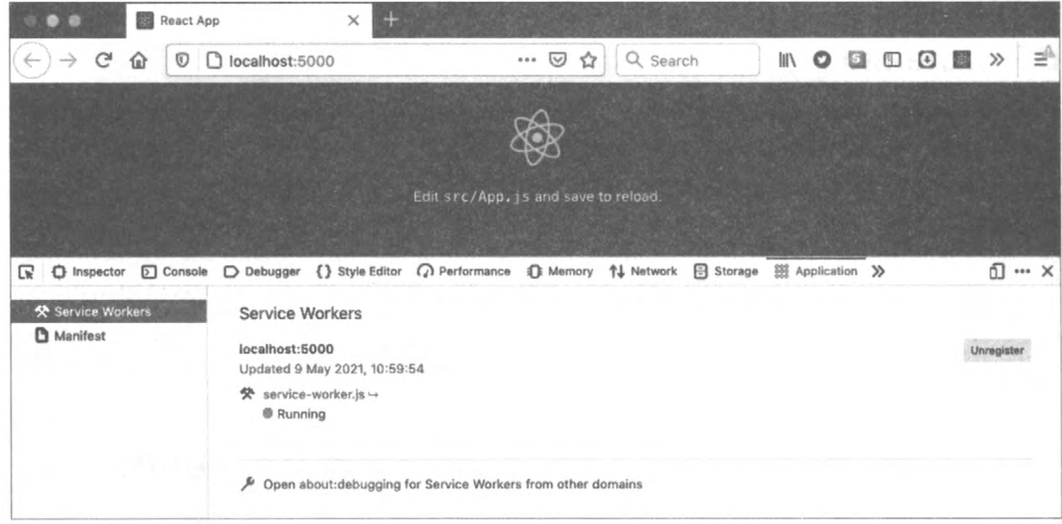


Рис. 11.14. Новый сервис-воркер начинает исполняться только после закрытия и повторного запуска приложения

Сервис-воркер может кешировать код приложения, поэтому если он не перестанет исполняться, то не загрузит самый последний код с сервера. Поэтому может случиться, что будет активна старая версия всего клиентского приложения. Чтобы

запустить новую версию приложения, нужно перезагрузить страницу (установить новый сервис-воркер), а затем закрыть и снова открыть вкладку (удалить старую версию сервис-воркера и запустить на исполнение новую).

Тестировщики быстро привыкают к этой слегка запутанной последовательности, чего нельзя сказать об обычных пользователях. В действительности то обстоятельство, что код обновится только на следующий раз, после того как станет доступным, обычно не представляет большой проблемы. Но это может вызвать трудности в случае значительных модификаций кода, например при обновлении API-интерфейса[6].

В некоторых случаях новый код требуется сразу же. Существует ли какой-либо способ немедленного удаления старых сервис-воркеров и обновления новой версии приложения?

РЕШЕНИЕ

Чтобы переключиться на новый сервис-воркер, требуется выполнить две операции: зарегистрировать сервис-воркер и перезагрузить страницу. Для приложений, созданных при помощи средства `create-react-app` или использующих код из шаблона `cra-template-pwa`[7], сервис-воркер регистрируется посредством функции `serviceWorkerRegistration.register`. Для этого может использоваться код в файле index.js, наподобие приведенного в листинге 11.20.

Листинг 11.20. Код файла index.js, используемый для регистрации сервис-воркера

```
import React from 'react'
import ReactDOM from 'react-dom'
import './index.css'
import App from './App'
import * as serviceWorkerRegistration from './serviceWorkerRegistration'
import reportWebVitals from './reportWebVitals'
ReactDOM.render(
  <React.StrictMode>
    <App />
  </React.StrictMode>,
  document.getElementById('root')
)
serviceWorkerRegistration.register()
reportWebVitals()
```

Даже созданный вручную код регистрации сервис-воркера, скорее всего, будет похожим на этот код.

[6] Этого не происходит при использовании семантического управления версиями конечных точек API-интерфейса.

[7] См. *раздел 11.2*.

Функции `serviceWorkerRegistration.register` можно передавать две функции обратного вызова, которые информируют об установке или обновлении сервис-воркера, как показано в листинге 11.21.

Листинг 11.21. Передача параметров функции serviceWorkerRegistration.register

```
serviceWorkerRegistration.register({
   onInstall: (registration) => {},
   onUpdate: (registration) => {},
})
```

Эти функции обратного вызова принимают в качестве параметра объект *регистрации*, который представляет собой обертку для сервис-воркера, только что установленного или обновленного браузером.

Установка сервис-воркера происходит при его загрузке. Но если старый сервис-воркер продолжает исполняться, то новый воркер будет в режиме ожидания до тех пор, пока не прекратится исполнение старого. Если сервис-воркер находится в режиме ожидания, вызывается функция `onUpdate`.

Мы хотим автоматически удалять старый сервис-воркер всякий раз при вызове функции `onUpdate`. Это позволит новому сервис-воркеру начать исполняться.

Сервис-воркер — это специальный тип веб-воркера. Веб-воркер представляет собой фрагмент кода JavaScript, который выполняется в отдельном потоке от кода JavaScript, исполняющегося в веб-странице. Взаимодействие с веб-воркерами осуществляется посредством передачи им асинхронных сообщений. Сервис-воркеры могут перехватывать сетевые запросы, поскольку браузер преобразует эти запросы в сообщения.

Таким образом, можно дать указание сервис-воркеру исполнить любой фрагмент кода, отправив ему сообщение. Чтобы сервис-воркер мог отвечать на сообщения, его можно снабдить слушателем событий получения сообщений:

```
self.addEventListener('message', (event) => {
   // Сюда вставляется код обработки сообщений
})
```

Переменная `self` содержит глобальный контекст для сервис-воркера (наподобие переменой `window` для кода страницы).

Код страницы может отправить сообщение новому сервис-воркеру, сообщая ему, что нужно выйти из режима ожидания и заменить старый сервис-воркер, как показано в листинге 11.22.

Листинг 11.22. Отправка сообщения сервис-воркеру

```
serviceWorkerRegistration.register({
   onUpdate: (registration) => {
      registration.waiting.postMessage({ type: 'SKIP_WAITING' })
   },
})
```

Код `registration.wating` — это ссылка на сервис-воркера, а код `registration.waiting.postMessage` отправляет воркеру сообщение.

После установки браузером новой версии сервис-воркера, но со старой версией воркера все еще исполняющейся, код приложения отправляет новому воркеру сообщение `SKIP_WAITING`.

Сервис-воркеры оснащены встроенной функцией `skipWating`, которая удаляет старый воркер, и позволяет новому начать исполняться. Таким образом, мы можем вызвать эту функцию в сервис-воркере, когда он получит сообщение `SKIP_WAITING`, как показано в листинге 11.23.

Листинг 11.23. Вызов функции `skipWating` в сервис-воркере

```
self.addEventListener('message', (event) => {
   if (event.data && event.data.type === 'SKIP_WAITING') {
      self.skipWaiting()
   }
})
```

Если теперь обновить приложение, то новый сервис-воркер сразу же заменит старый.

Теперь осталось выполнить последний шаг: нужно перезагрузить страницу, чтобы можно было загрузить новый код приложения через новый сервис-воркер. Это означает, что обновленная версия файла index.js приложения будет иметь содержимое, приведенное в листинге 11.24.

Листинг 11.24. Содержимое обновленного файла index.js

```
import React from 'react'
import ReactDOM from 'react-dom'
import './index.css'
import App from './App'
import * as serviceWorkerRegistration from './serviceWorkerRegistration'
import reportWebVitals from './reportWebVitals'
ReactDOM.render(
   <React.StrictMode>
      <App />
   </React.StrictMode>,
   document.getElementById('root')
)
serviceWorkerRegistration.register({
   onUpdate: (registration) => {
      registration.waiting.postMessage({ type: 'SKIP_WAITING' })
      window.location.reload()
   },
})
reportWebVitals()
```

После установки этой новой версии кода приложение будет автоматически обновляться при каждом его изменении. Теперь на вкладке **Application** окна инструментов разработчика вместо работающего старого сервис-воркера и ожидающего своей очереди нового будет отображаться только работающий новый воркер (рис. 11.15).

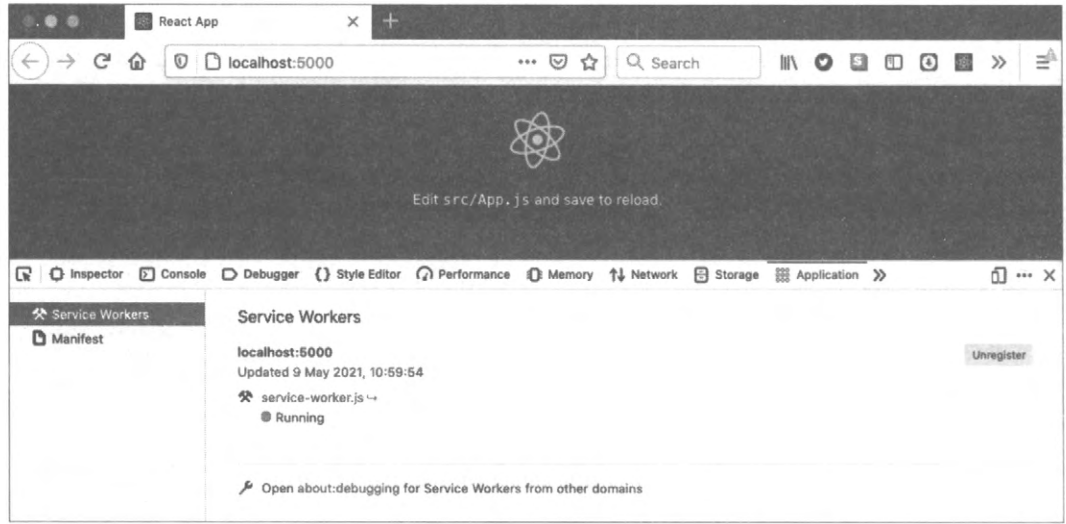


Рис. 11.15. Теперь новый сервис-воркер сразу же заменяет старый

Обсуждение

Перезагрузка страницы вызывает "моргание" страницы при загрузке нового кода. В случае большого приложения это может вызывать неприятное восприятие у пользователя, поэтому целесообразно спросить у пользователя, хочет ли он выполнить обновление приложения, прежде чем перезагружать его. Этот подход применяется компанией Gmail для всех значительных обновлений.

Исходный код данного рецепта можно загрузить на веб-сайте GitHub по адресу **https://oreil.ly/Lbal7**.

11.5. Добавление извещений

ЗАДАЧА

Одно из преимуществ сервис-воркеров в частности и веб-воркеров в целом состоит в том, что они не прекращают исполняться, даже если пользователь ушел со страницы. Воркер будет продолжать исполняться в фоновом режиме, пока работает браузер. Поэтому в случае медленного сервис-воркера можно уйти со страницы или даже закрыть вкладку, но воркер будет продолжать исполняться, пока не выполнит свою задачу.

Но как быть, если пользователь хочет знать, завершилось ли наконец выполнение фоновой задачи? Ведь сервис-воркеры не имеют никакого визуального интерфейса. Хотя они могут управлять веб-страницами, но не способны обновлять их. Веб-страница и сервис-воркер могут взаимодействовать посредством сообщений.

С учетом отсутствия у сервис-воркеров визуального интерфейса, как они могут сообщать пользователю, когда произойдет что-либо важное?

РЕШЕНИЕ

Для решения данной задачи мы создадим возможность получения уведомлений от сервис-воркера. По нажатии кнопки наше демонстрационное приложение (рис. 11.16) начнет исполнять длительный процесс, занимающий около 20 секунд до завершения.

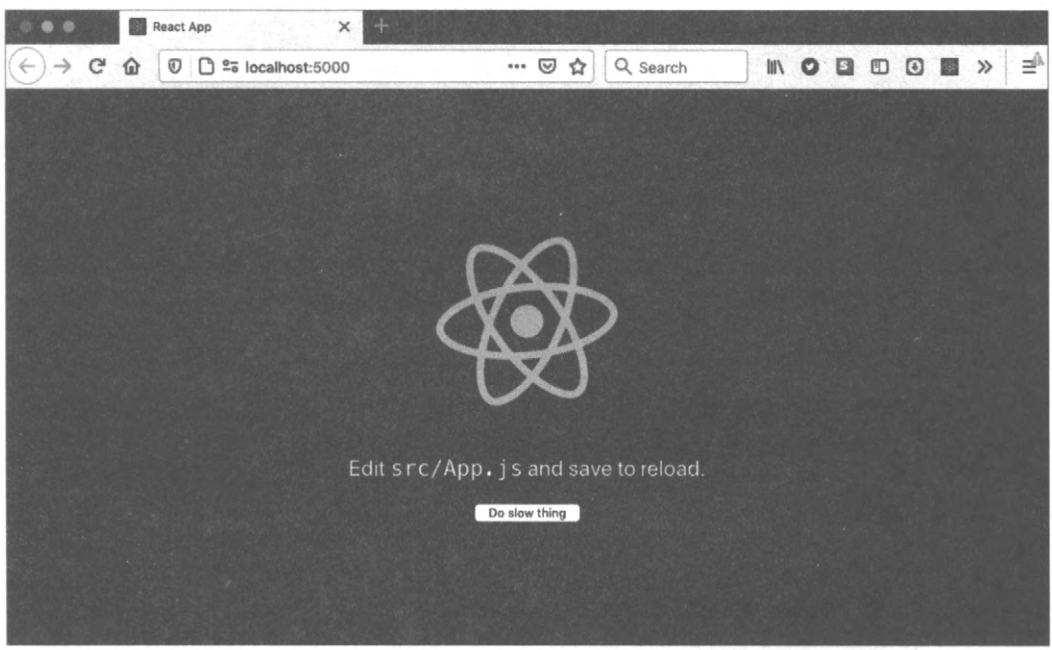


Рис. 11.16. Нажатие кнопки **Do slow thing** запускает на исполнение длительный процесс

Чтобы приложение смогло отправлять уведомление о завершении этого процесса, пользователь должен предоставить соответствующее разрешение (рис. 11.17). Если отказать в таком разрешении, то задача все равно будет исполняться в фоновом режиме, но пользователь не будет уведомлен о ее завершении.

Извещения пользуются дурной славой. Обычно веб-сайт отправляет их пользователю в виде непрошеных рекламных сообщений. В целом лучше не спешить спрашивать у пользователя разрешение, пока ему не будет очевидно, зачем оно требуется. Избегайте спрашивать разрешение для отправки сообщений сразу же после загрузки страницы, поскольку пользователь будет без понятия, почему вы хотите отправлять их.

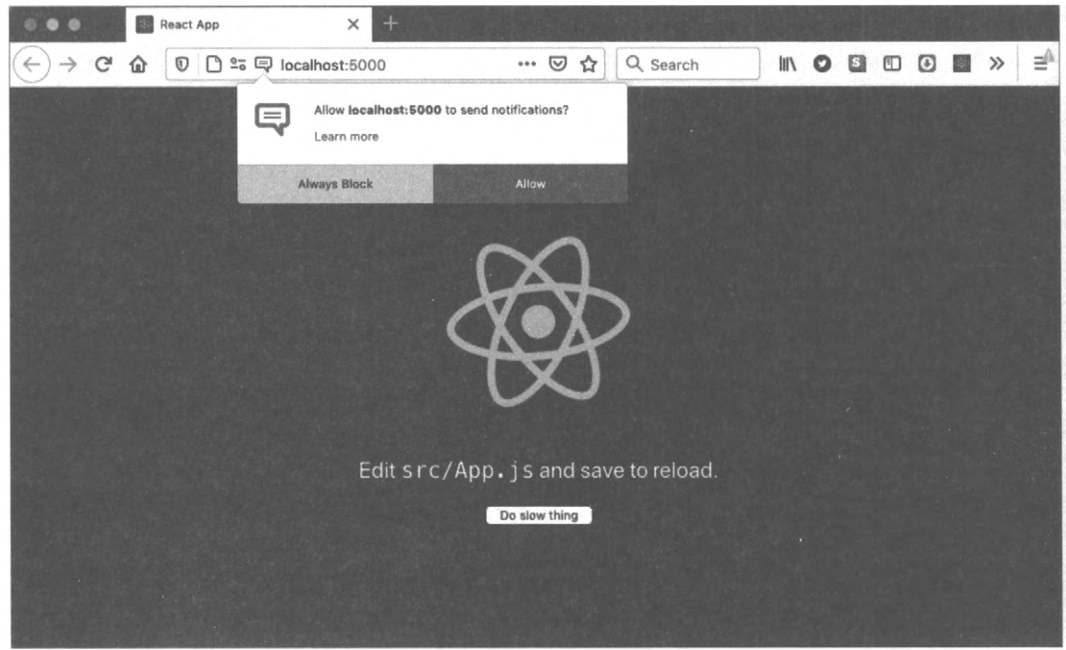


Рис. 11.17. Для получения уведомлений о завершении исполнения задачи нужно разрешить их

Сервис-воркер начнет исполнять некий код, который просто делает паузу длительностью в 20 секунд, после чего отображается уведомление о завершении его работы (рис. 11.18).

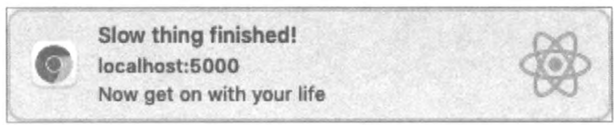


Рис. 11.18. Уведомление о завершении исполнения задачи

Начнем разбираться, что представляет собой этот код. В компонент `App` мы добавляем кнопку для запуска фоновой задачи, но при этом делаем ее видимой, лишь удостоверившись, что браузер поддерживает сервис-воркеры (листинг 11.25).

Листинг 11.25. Код кнопки запуска задачи

```
function App() {
   const startTask = () => {
      // Код для запуска задачи
   }
   return (
      <div className="App">
         <header className="App-header">
            <img src={logo} className="App-logo" alt="logo" />
```

```
          <p>
            Edit <code>src/App.js</code> and save to reload.
          </p>
          {'serviceWorker' in navigator && (
            <button onClick={startTask}>Do slow thing</button>
          )}
        </header>
      </div>
    )
}
```

Нажатие пользователем этой кнопки вызывает функцию `startTask`. В ней можно отображать сообщение с запросом разрешения показывать уведомления (листинг 11.26).

Листинг 11.26. Код функции `startTask`

```
const startTask = () => {
  Notification.requestPermission((permission) => {
    navigator.serviceWorker.ready.then(() => {
      const notifyMe = permission === 'granted'
      // Здесь начинается код задачи
    })
  })
}
```

Если пользователь дает разрешение, то строке `permission` присваивается значение `granted`, в результате чего константа `notifyMe` получает значение `true`. Мы можем исполнять задачу в сервис-воркере и сообщить ему, можно ли ему отправлять уведомление о завершении выполнения этой задачи.

Но мы не можем общаться с сервис-воркерами напрямую. Вместо этого нам нужно отправлять сообщения, поскольку сервис-воркеры исполняются в отдельном потоке от кода веб-страницы.

Текущего сервис-воркера, управляющего страницей, можно определить при помощи свойства `navigator.service Worker.controller`. Теперь мы можем отправить этому сервис-воркеру сообщение, как показано в листинге 11.27.

Листинг 11.27. Отправка сообщения сервис-воркеру

```
const startTask = () => {
  Notification.requestPermission((permission) => {
    navigator.serviceWorker.ready.then(() => {
      const notifyMe = permission === 'granted'
      navigator.serviceWorker.controller.postMessage({
        type: 'DO_SLOW_THING',
```

```
            notifyMe,
         })
      })
   })
}
```

В нашем демонстрационном сообщении код сервис-воркера находится в файле service-worker.js. Его можно оснастить возможностью получать сообщения, добавив в него код обработчика события получения сообщения, показанного в листинге 11.28.

Листинг 11.28. Код обработчика события получения сообщения

```
self.addEventListener('message', (event) => {
   ...
   if (event.data && event.data.type === 'DO_SLOW_THING') {
      // Сюда вставляется код для выполнения длительной задачи
   }
})
```

В контексте сервис-воркера выражение `self` обозначает объект с глобальной областью видимости. Оно эквивалентно выражению `window` в коде веб-страницы. Следующим шагом эмулируем длительную задачу, вызывая функцию `setTimeout`, которая просто ожидает в течение 20 секунд, после чего отправляет сообщение на консоль JavaScript в окне инструментов разработчика (листинг 11.19)[8].

Листинг 11.29. Код эмулирования длительной задачи

```
self.addEventListener('message', (event) => {
   ...
   if (event.data && event.data.type === 'DO_SLOW_THING') {
      setTimeout(() => {
         console.log('Slow thing finished!')
         // TODO: Send notification here
      }, 20000)
   }
})
```

Все, что осталось теперь сделать, — отобразить уведомление. Это можно реализовать при помощи метода `showNotification` объекта `registration` сервис-воркера, как показано в листинге 11.30.

[8] Сообщение сможет отобразиться только в браузере Chrome. В браузере Firefox оно не будет отображаться, поскольку этот браузер не разрешает сервис-воркерам доступ к консоли JavaScript.

Листинг 11.30. Код для отображения уведомления

```
self.addEventListener('message', (event) => {
  ...
  if (event.data && event.data.type === 'DO_SLOW_THING') {
    setTimeout(() => {
      console.log('Slow thing finished!')
      if (event.data.notifyMe) {
        self.registration.showNotification('Slow thing finished!', {
          body: 'Now get on with your life',
          icon: '/logo512.png',
          vibrate: [100, 100, 100, 200, 200, 200, 100, 100, 100],
          // tag: 'some-id-if-you-do-not-want-duplicates'
        })
      }
    }, 20000)
  }
})
```

Обратите внимание на то, что прежде чем отображать уведомление, мы проверяем значение константы `event.data.notifyMe`, которую добавили в сообщение в коде веб-страницы.

Уведомление принимает в качестве параметров заголовок и объект опций. Опции позволяют модифицировать поведение уведомления. В данном случае опции состоят из текста `body`, значка `icon` и последовательности вибраций `vibrate`. Если устройство пользователя поддерживает опцию вибрации, то при отображении уведомления устройство также должно вибрировать последовательность точка-точка-точка-тире-тире-тире-точка-точка-точка.

Существует также и опция метки `tag`, которую мы закомментировали в листинге 11.30. Опция метки служит для однозначной идентификации уведомления, чтобы не допустить многократной отправки пользователю одного и того же уведомления. Если эту опцию отключить, то каждый вызов функции `showNotification` будет приводить к отображению уведомления.

Чтобы испытать наш код в действии, нужно сначала выполнить сборку приложения, поскольку сервис-воркеры могут исполняться только в рабочей версии приложения:

```
$ yarn run build
```

Далее нам необходимо запустить сервер на содержимом сгенерированной папки build. Для этого установим модуль `serve`, а затем запустим его на исполнение:

```
$ serve -s build
```

Теперь откроем приложение по адресу **http://localhost:5000** и запустим длительную задачу, нажав соответствующую кнопку. После этого можно перейти на какую-либо другую страницу или вообще закрыть вкладку приложения, но запущенная

длительная задача будет продолжать исполняться. Остановить ее до истечения времени тайм-аута можно, только закрыв браузер.

По истечении 20 секунд после запуска задачи в браузере должно отобразиться уведомление наподобие показанного на рис. 11.19.

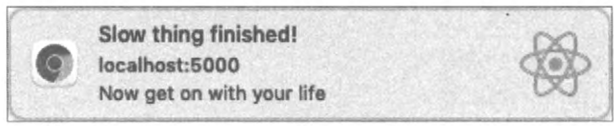


Рис. 11.19. Уведомление о завершении длительной задачи, отображаемое на компьютере Mac

Наверное, многим хотелось бы исполнить это приложение на мобильном устройстве, чтобы проверить работу вибрации в уведомлении. Касательно этого следует иметь в виду, что сервис-воркеры активируются только при обращении к серверу на *localhost* или через протокол HTTPS. Чтобы проверить приложение через протокол HTTPS, см. *раздел 7.3* для запуска его на сервере.

С учетом того, что уведомления могут отображаться после закрытия соответствующей страницы, целесообразно предоставить пользователю простой способ возврата обратно к приложению. Это можно реализовать, добавив в наш сервис-воркер обработчик события щелчка по уведомлению. Тогда при щелчке по уведомлению браузер отправит сервис-воркеру событие `notificationclick`, которое можно обработать при помощи обработчика, приведеного в листинге 11.31.

Листинг 11.31. Обработчик события `notificationclick`

```
self.addEventListener('notificationclick', (event) => {
  event.notification.close()
  // ЗАДАЧА Возвращаемся обратно к приложению
})
```

Уведомление можно закрыть, вызвав метод `event.notification.close`. Но как отправить пользователя обратно к приложению React?

Сервис-воркер может управлять несколькими вкладками браузера, которые называются его *клиентами*. Это вкладки, сетевые запросы которых перехватываются сервис-воркером. Список клиентов сервис-воркера можно получить из его объекта `self.clients`. Посредством метода `openWindow` этого объекта можно открыть новую вкладку в браузере, как показано в листинге 11.32.

Листинг 11.32. Открытие новой вкладки посредством метода `openWindow`

```
self.addEventListener('notificationclick', (event) => {
  event.notification.close()
  if (self.clients.openWindow) {
    self.clients.openWindow('/')
  }
})
```

Теперь при щелчке мышью по уведомлению браузер откроет домашнюю страницу приложения React.

Но мы можем сделать нечто лучшее. Если пользователь переключился на другую вкладку браузера, но приложение React продолжает исполняться, мы можем перевести фокус на вкладку, отправившую уведомление.

Для этого нам нужно получить массив всех вкладок, управляемых нашим сервис-воркером, и посмотреть, нет ли среди них с совпадающим путем. Если есть, то мы переключаем фокус на эту вкладку, как показано в листинге 11.33.

Листинг 11.33. Переключение фокуса на вкладку, издавшую уведомление

```
self.addEventListener('notificationclick', (event) => {
  event.notification.close()
  event.waitUntil(
    self.clients
      .matchAll({
        type: 'window',
      })
      .then((clientList) => {
        const returnPath = '/'
        const tab = clientList.find((t) => {
          return t.url === self.location.origin + returnPath
        })
        if (tab && 'focus' in tab) {
          tab.focus()
        } else if (self.clients.openWindow) {
          self.clients.openWindow(returnPath)
        }
      })
  )
})
```

Щелчок по уведомлению возвращает нас на уже открытую вкладку, а не создает новую (рис. 11.20).

Обсуждение

Уведомления предоставляют отличный способ для информирования пользователя о важных событиях. Критическим аспектом использования уведомлений является разъяснение пользователю, почему он должен согласиться на получение уведомлений, а затем отправлять их только в случае важных событий.

Исходный код данного рецепта можно загрузить на веб-сайте GitHub по адресу **https://oreil.ly/ZkcrR**.

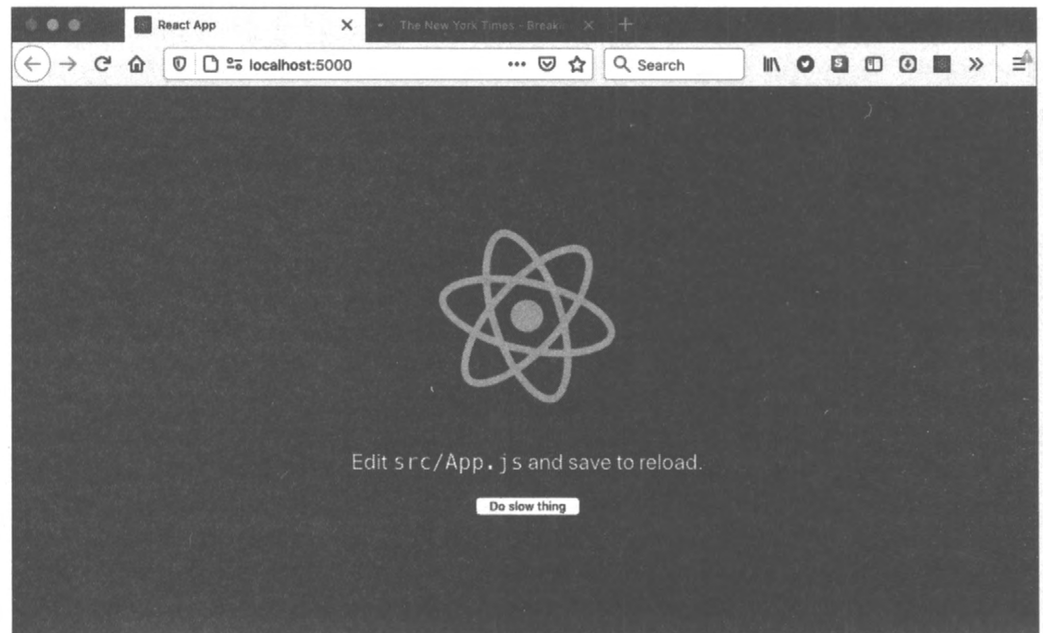


Рис. 11.20. Щелчок по уведомлению переключает фокус на вкладку приложения, если она все еще открыта

11.6. Модификации в режиме офлайн посредством фоновой синхронизации

ЗАДАЧА

Предположим, что ваше приложение используется там, где отсутствует подключение к сети, например, в поезде метро[9]. Предварительное кеширование кода означает, что в таком случае с открытием приложения не будет возникать никаких проблем. Также пользователь сможет перемещаться между страницами, и все будет выглядеть работающим должным образом.

Но что произойдет, если пользователь выполнит какую-либо операцию, отправляющую данные на сервер? Например, что если он попытается отправить сообщение?

РЕШЕНИЕ

Фоновая синхронизация — это способ постановки в очередь сетевых запросов при недоступности сервера с последующей их отправкой в дальнейшем, когда сервер становится доступным.

[9] Надо сказать, что все больше и больше метрополитенов оснащаются мобильными ретрансляционными станциями.

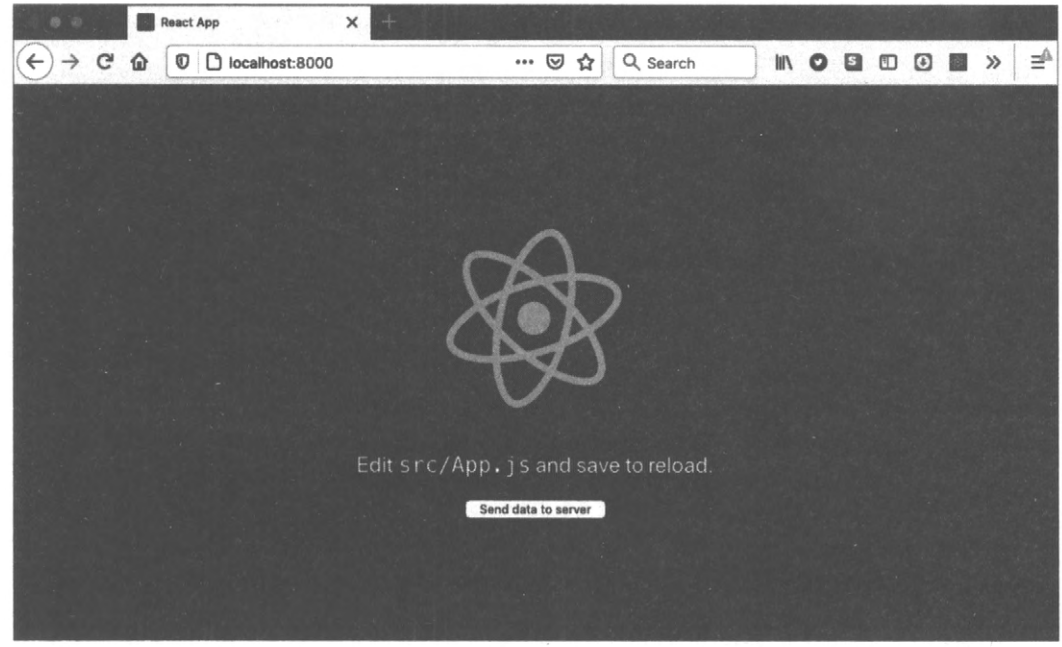


Рис. 11.21. При нажатии пользователем кнопки **Send data to server** демонстрационное приложение отравляет данные на сервер

Наше демонстрационное приложение будет отправлять данные на сервер бэкенда при нажатии пользователем соответствующей кнопки (рис. 11.21).

Прежде чем запускать приложение, нужно выполнить его сборку:

```
$ yarn run build
```

Наш демонстрационный проект содержит сервер в файле server/index.js (листинг 11.34).

Листинг 11.34. Код сервера для проекта

```
const express = require('express')
const app = express()
app.use(express.json())
app.use(express.static('build'))
app.post('/endpoint', (request, response) => {
  console.log('Server received data', request.body)
  response.send('OK')
})
app.listen(8000, () => console.log('Launched on port 8000''))
```

Сервер раздает содержимое из папки `build`, в которую публикуется сгенерированный код. Он также отображает данные запросов `POST`, отправляемых по адресу **http://localhost:8000/endpoint**.

Сервер запускается на исполнение следующей командой:

```
$ node server
```

Далее откроем в браузере страницу по адресу **http://localhost:8000** и несколько раз нажмем кнопку для отправки данных на сервер. В окне сервера должны отображаться данные, показанные в листинге 11.35.

Листинг 11.35. Данные, отправляемые на сервер бэкенда

```
$ node server
Launched on port 8000!
Server received data { timeIs: '2021-05-09T18:59:37.280Z' }
Server received data { timeIs: '2021-05-09T18:59:37.720Z' }
Server received data { timeIs: '2021-05-09T18:59:38.064Z' }
Server received data { timeIs: '2021-05-09T18:59:38.352Z' }
```

В листинге 11.36 приведен код, который отправляет эти данные на сервер. В нем используется функция `fetch`, которая при нажатии кнопки отправляет на сервер текущее время методом `POST`.

Листинг 11.36. Код для отправки данных на сервер

```
import React from 'react'
import logo from './logo.svg'
import './App.css'
function App() {
  const sendData = () => {
    const options = {
      method: 'POST',
      body: JSON.stringify({ timeIs: new Date() }),
      headers: {
        'Content-Type': 'application/json',
      },
    }
    fetch('/endpoint', options)
  }
  return (
    <div className="App">
      <header className="App-header">
        <img src={logo} className="App-logo" alt="logo" />
        <p>
          Edit <code>src/App.js</code> and save to reload.
        </p>
        <button onClick={sendData}>Send data to server</button>
      </header>
    </div>
  )
}
export default App
```

Если теперь остановить сервер, то нажатие кнопки будет генерировать последовательность сообщений об ошибке сетевого запроса, как показано на рис. 11.22.

Остановка сервера эмулирует ситуацию, когда пользователь временно теряет подключение к сети и пытается отправить данные из приложения.

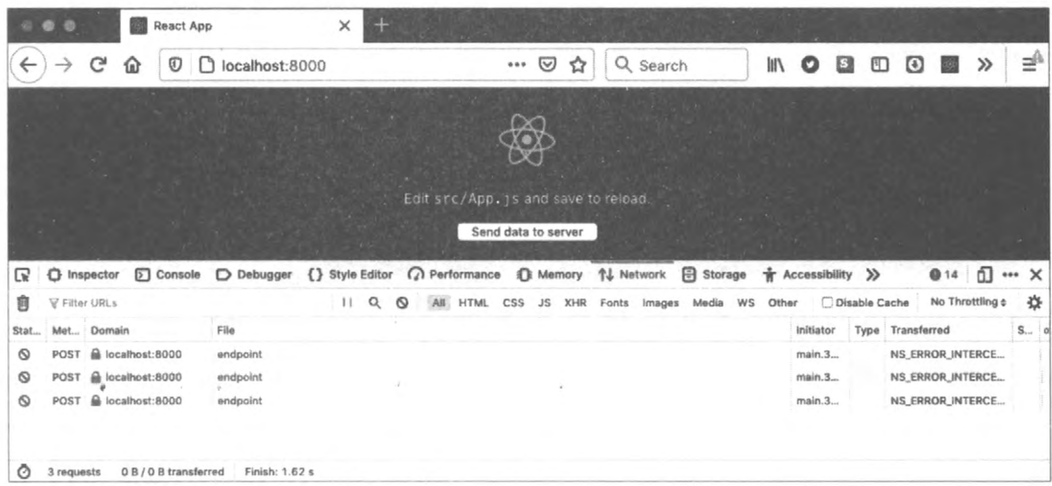


Рис. 11.22. При отсутствии сервера бэкенда сетевые запросы завершаются неудачно

Указанную проблему можно исправить при помощи сервис-воркеров. Сервис-воркер способен перехватывать сетевые запросы, создаваемые веб-страницей прогрессивного веб-приложения. В предыдущих рецептах этой главы мы рассмотрели использование сервис-воркеров для обработки проблем с подключением к сети с помощью возврата версии файлов из локального кеша. В нынешней ситуации нам нужно обрабатывать данные, идущие в противоположном направлении: от браузера к серверу.

Нам требуется сохранить в кеше запросы `POST`, которые приложение пытается передать на сервер, а после восстановления контакта с сервером отравить ему эти запросы.

Для этого мы воспользуемся библиотекой `workbox-background-sync`. Фоновая синхронизация представляет собой API-интерфейс для перенаправления сетевых запросов в очередь в тех случаях, когда отсутствует связь с сервером. Это сложный API-интерфейс, и не все браузеры поддерживают его.

Библиотека `workbox-background-sync` значительно облегчает работу с API-интерфейсом фоновой синхронизации, а также позволяет применять его на браузерах, которые не поддерживают его самостоятельно, например Firefox.

В нашем демонстрационном приложении код сервис-воркера находится в файле service-worker.js. Чтобы реализовать в сервис-воркере возможность фоновой синхронизации, добавим в этот файл код, приведенный в листинге 11.37.

Листинг 11.37. Код для добавления возможности фоновой синхронизации

```
import { NetworkOnly } from 'workbox-strategies'
import { BackgroundSyncPlugin } from 'workbox-background-sync'
// Сюда вставляется прочий код сервис-воркера...
registerRoute(
   //endpoint/,
   new NetworkOnly({
      plugins: [
         new BackgroundSyncPlugin('endPointQueue1', {
            maxRetentionTime: 24 * 60,
         }),
      ],
   }),
   'POST'
)
```

Этот код регистрирует в сервис-воркере новый маршрут, указывая, как обрабатывать сетевые запросы по конкретным URL-адресам. В данном случае мы создаем маршрут для обработки всех запросов по адресу **http://localhost.8000/endpoint**. Сопоставление адресов заданному пути осуществляется с помощью регулярного выражения. Затем используется стратегия Network Only (**https://oreil.ly/rLqLq**), при которой браузер отправляет все запросы сервис-воркеру, а все ответы берутся из Сети. Но эта стратегия сконфигурирована для работы с подключаемым модулем фоновой синхронизации `BackgroundSyncPlugin`. Третий параметр в маршруте указывает, что интерес представляют только запросы `POST` к концевой точке.

Когда приложение отправляет запрос `POST` по адресу **http://localhost:8000/endpoint**, сервис-воркер перехватывает его. Далее сервис-воркер перенаправляет перехваченный запрос на сервер и при успешной отправке возвращает полученный ответ веб-странице. В случае недоступности сервера сервис-воркер возвращает веб-странице сообщение об ошибке, а затем добавляет запрос в очередь `endPointQueue1`, чтобы отправить его повторно.

Библиотека Workbox сохраняет очереди в индексируемых базах данных в браузере. Присвоенное переменной `maxRetentionTime` значение `24 * 60` сохраняет запросы в базе данных вплоть до 24 часов.

Библиотека `workbox-background-sync` повторно отправляет поставленные в очередь запросы при любой возможности, когда она полагает, что сервер может быть доступным, например при возобновлении сетевого подключения. Также попытки повторной отправки предпринимаются каждые несколько минут.

Если теперь перезапустить сервер и подождать около пяти минут, мы должны увидеть, что неудачно завершенные запросы отображаются в окне сервера (листинг 11.38).

Листинг 11.38. Отображение в сервере запросов, ранее завершившихся ошибкой

```
$ node server
Launched on port 8000!
Server received data { timeIs: '2021-05-09T21:26:11.068Z' }
Server received data { timeIs: '2021-05-09T21:02:44.647Z' }
Server received data { timeIs: '2021-05-09T21:02:45.647Z' }
```

Браузер Chrome можно заставить отправлять запросы немедленно. Для этого в панели инструментов разработчика откройте вкладку **Application**, выберите в ней требуемый сервис-воркер, а затем отправьте сообщение синхронизации по адресу `workbox-background-sync:endPointQueue1`, как показано на рис. 11.23.

Рис. 11.23. Принудительная синхронизация в браузере Chrome

Обсуждение

Фоновая синхронизация — чрезвычайно мощная возможность, но ее реализацию следует хорошо продумать. Порядок, в котором код клиента отправляет запросы, не обязательно будет совпадать с порядком, в котором эти запросы обрабатываются сервером.

Точный порядок, скорее всего, не будет иметь значения в случае создания посредством запросов `POST` простого набора ресурсов. Например, при покупке книг в интернет-магазине точный порядок их приобретения не имеет значения.

Но при создании зависимых ресурсов или множественных обновлений одного и того же ресурса[10] нужно быть осторожным. Если, например, изменить номер кре-

[10] При использовании API-интерфейса на основе стиля REST обновления, скорее всего, осуществлялись бы посредством запроса PUT или PATCH.

дитной карточки на 1111 1111 1111 1111, а затем на 2222 2222 2222 2222, то в зависимости от порядка этих изменений конечный результат будет совершенно разным.

Исходный код данного рецепта можно загрузить на веб-сайте GitHub по адресу **https://oreil.ly/NFVAY**.

11.7. Добавляем специализированный установочный пользовательский интерфейс

ЗАДАЧА

Во многих отношениях прогрессивные веб-приложения ведут себя подобно приложениям, установленным локально. Их можно устанавливать вместе с другими приложениями на настольных компьютерах или мобильных устройствах. Многие браузеры позволяют создать ярлык приложения на текущем устройстве, чтобы его можно было запускать в отдельном окне. На настольных компьютерах есть возможность добавить ярлык в меню запуска. На мобильных устройствах ярлык приложения можно разместить на домашнем экране.

Но многие пользователи просто не знают о возможности устанавливать прогрессивные веб-приложения. К тому же зачастую в интерфейсе, используемом в браузерах для обозначения возможности установки таких приложений (рис. 11.24), трудно отыскать требуемую опцию.

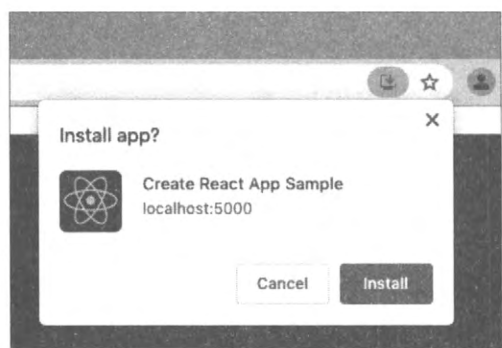


Рис. 11.24. Прогрессивные веб-приложения устанавливаются посредством небольшой кнопки в строке адреса браузера

Такой интерфейс объясняется стремлением разработчиков браузеров максимизировать область экрана мобильного устройства, доступного для веб-страницы. Но если есть основания полагать, что локальная установка приложения может быть полезной для его пользователей, то его можно снабдить специальным установочным пользовательским интерфейсом. Но как именно это можно сделать?

РЕШЕНИЕ

Некоторые браузеры[11] генерируют событие `beforeinstallprompt` при обнаружении полноценного прогрессивного веб-приложения[12].

Это событие можно захватить и использовать для отображения вашего специального установочного пользовательского интерфейса. Начнем с создания файла MyInstaller.js, содержащего код компонента `MyInstaller`, приведенный в листинге 11.39.

Листинг 11.39. Код файла MyInstaller.js

```
import React, { useEffect, useState } from 'react'
const MyInstaller = ({ children }) => {
   const [installEvent, setInstallEvent] = useState()
   useEffect(() => {
      window.addEventListener('beforeinstallprompt', (event) => {
         event.preventDefault()
         setInstallEvent(event)
      })
   }, [])
   return (
      <>
         {installEvent && (
            <button
               onClick={async () => {
                  installEvent.prompt()
                  await installEvent.userChoice
                  setInstallEvent(null)
               }}
            >
               Install this app!
            </button>
         )}
         {children}
      </>
   )
}
export default MyInstaller
```

Компонент `MyInstaller` выполняет захват события `onbeforeinstallprompt` и сохраняет его в переменной `installEvent`. Затем при наличии этого события отображается спе-

[11] На момент подготовки материала данной книги это событие поддерживали браузеры Chrome, Edge и Samsung Internet.

[12] Проверить, удовлетворяет ли ваше приложение требованиям прогрессивного веб-приложения, можно при помощи инструмента Lighthouse из набора инструментов разработчика Chrome. Этот инструмент не только определяет, отвечает ли приложение заданным требованиям, но также в случае их неудовлетворения укажет соответствующие причины.

циальный пользовательский интерфейс, который в данном случае состоит из простой кнопки. Далее этот компонент можно вставить в разрабатываемое приложение, например, как показано в листинге 11.40.

Листинг 11.40. Пример вставки компонента `MyInstaller` в приложение

```
function App() {
   return (
      <div className="App">
         <MyInstaller>
            <header className="App-header">
               <img src={logo} className="App-logo" alt="logo" />
               <p>
                  Edit <code>src/App.js</code> and save to reload.
               </p>
               <a
                  className="App-link"
                  href="https://reactjs.org"
                  target="_blank"
                  rel="noopener noreferrer"
               >
                  Learn React
               </a>
            </header>
         </MyInstaller>
      </div>
   )
}
```

Теперь выполним сборку приложения и запустим его на исполнение:

```
$ yarn run build
$ serve -s build
```

В результате в верхней части окна приложения должна отображаться кнопка для его установки **Install this app!** (рис. 11.25).

Данная кнопка не будет отображаться при исполнении приложения с использованием сервера разработки:

```
$ yarn run start
```

Это объясняется тем, что приложение считается прогрессивным веб-приложением только при наличии исполняющегося сервис-воркера, а сервис-воркеры могут функционировать только в рабочем режиме.

При нажатии кнопки установки приложения компонент `MyInstaller` исполняет метод `Event.prompt`, который отображает обычное диалоговое окно установки (рис. 11.26).

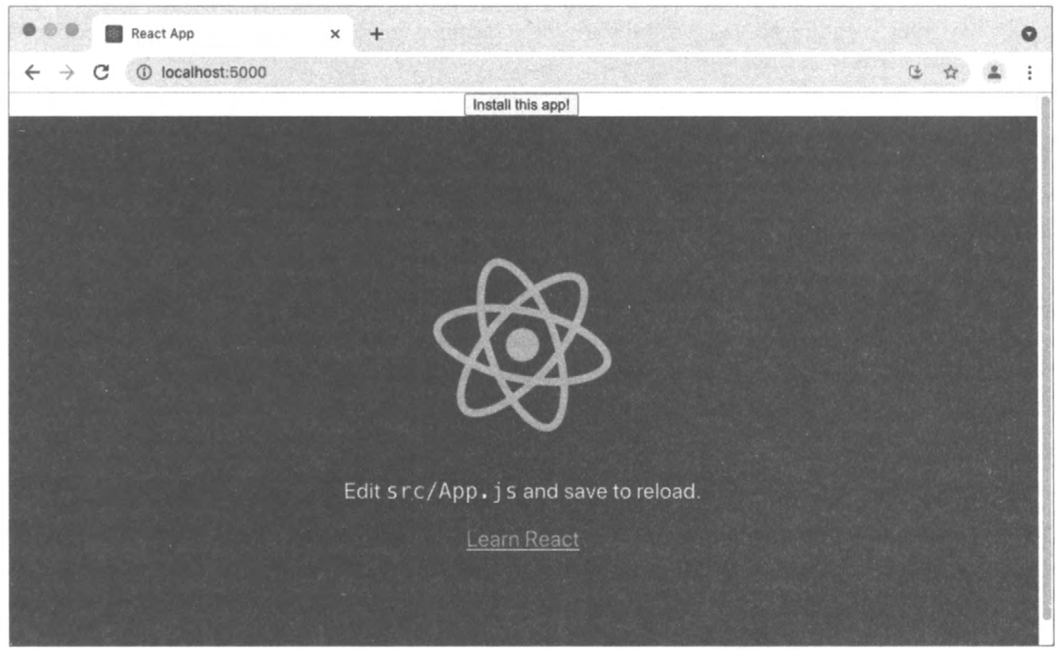


Рис. 11.25. Кнопка установки приложения отображается посередине вверху окна приложения

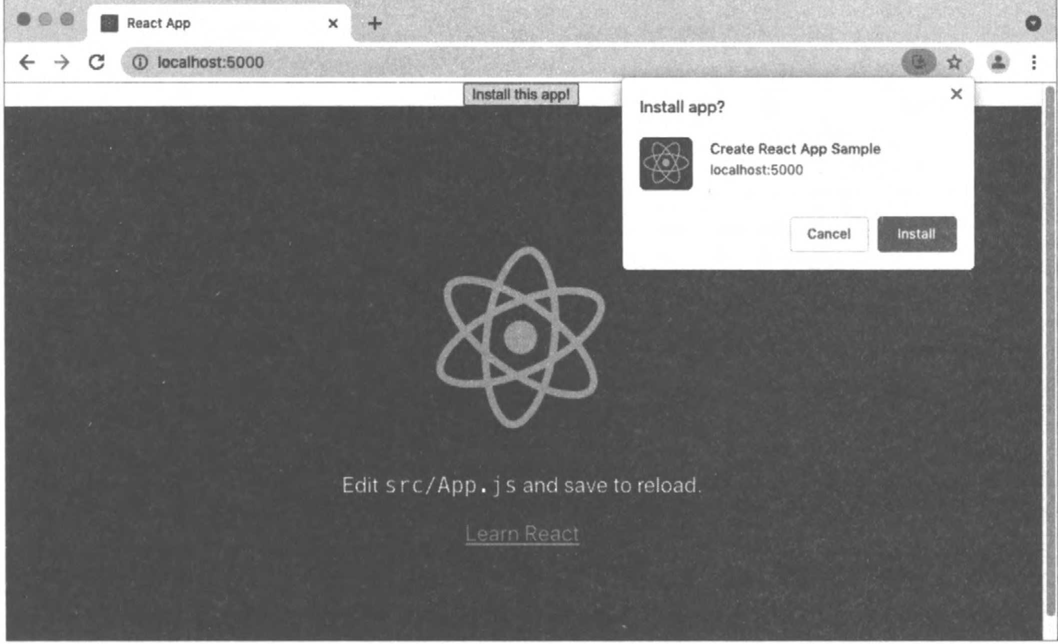


Рис. 11.26. Нажатие кнопки установки приложения открывает диалоговое окно установки

Если устройство уже установило приложение, то браузер не активирует событие `onbeforeinstallprompt`.

Если пользователь решит установить приложение, запускается отдельное окно приложения[13]. На настольном компьютере может открыться окно проводника Windows (Finder в компьютере Mac), содержащее значок запуска приложения, которое можно добавить в меню запуска компьютера (рис. 11.27). На мобильных устройствах этот значок будет размещен на домашнем экране.

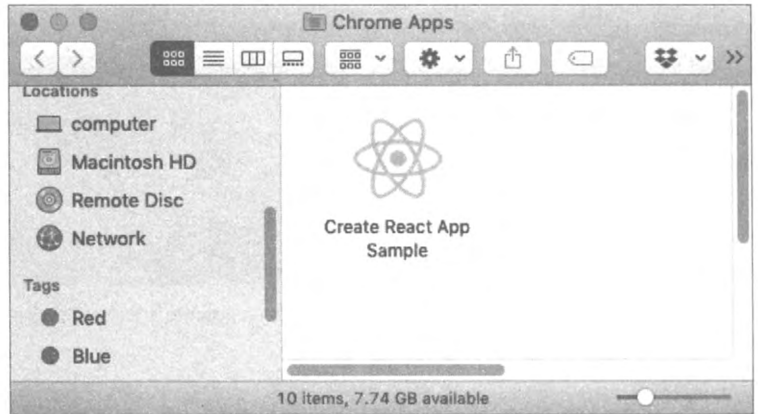


Рис. 11.27. Браузер создает значок запуска для приложения

Обсуждение

Локальная установка — это замечательная возможность для людей, которые часто пользуются приложением. Судя по нашему опыту, многие пользователи не знают, что для некоторых сайтов доступна опция локальной установки, поэтому целесообразно добавить в приложение специальный интерфейс для его установки. Но при этом следует остерегаться, чтобы не создать назойливый интерфейс, если имеются основания полагать, что вероятные посетители вашего сайта будут случайными. Лучше всего не запускать окно запроса установки автоматически при загрузке страницы, поскольку это, скорее всего, будет вызывать раздражение у пользователей и они больше не вернутся на ваш веб-сайт.

Исходный код данного рецепта можно загрузить на веб-сайте GitHub по адресу **https://oreil.ly/Dbmpc**.

---

[13] Тогда как приложение создает отдельное окно, это окно удаляется при закрытии веб-браузера. При непосредственном запуске приложения оно также запускает и веб-браузер, если он еще не запущен.

11.8. Предоставление ответов в режиме офлайн

ЗАДАЧА

Кешировать в приложении все сторонние ресурсы нежелательно, поскольку для этого потребуется слишком много места. Это означает, что иногда ваш код не сможет загрузить все требуемые ресурсы. Например, на рис. 11.28 мы видим, как приложение, которое мы создали в одной из предыдущих глав, отобразило несколько изображений, взятых со стороннего сайта.

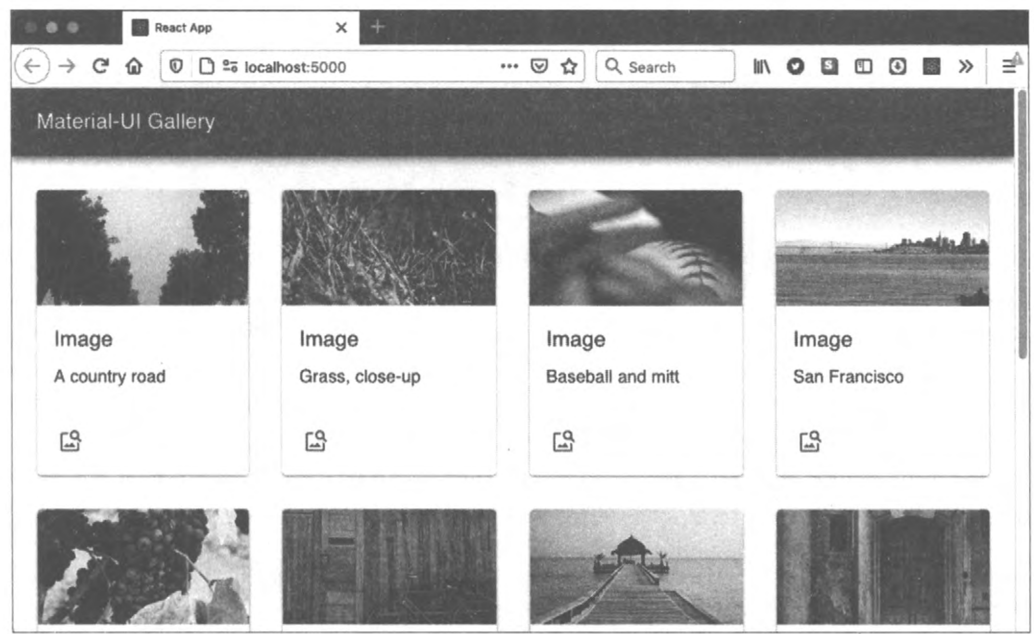


Рис. 11.28. Приложение отображает изображения, взятые с сайта **http://picsum.photos**

Для работы в автономном режиме можно при помощи сервис-воркера сохранить в локальном кеше весь код приложения. Но при этом сохранять так сами изображения со стороннего веб-сайта нежелательно вследствие слишком большого их количества. Это означает, что при отсутствии сетевого подключения работа приложения продолжится, но изображения будут отсутствовать (рис. 11.29).

Было бы неплохо заменить отсутствующие сторонние изображения какой-либо локальной версией. Таким образом, при работе в автономном режиме пользователь все равно будет видеть хоть какое-то изображение.

Это частный случай общей задачи, в которой желательно заменить заглушкой недоступный внешний файл большого размера, например видео- или аудиофайл или даже полностью веб-страницу.

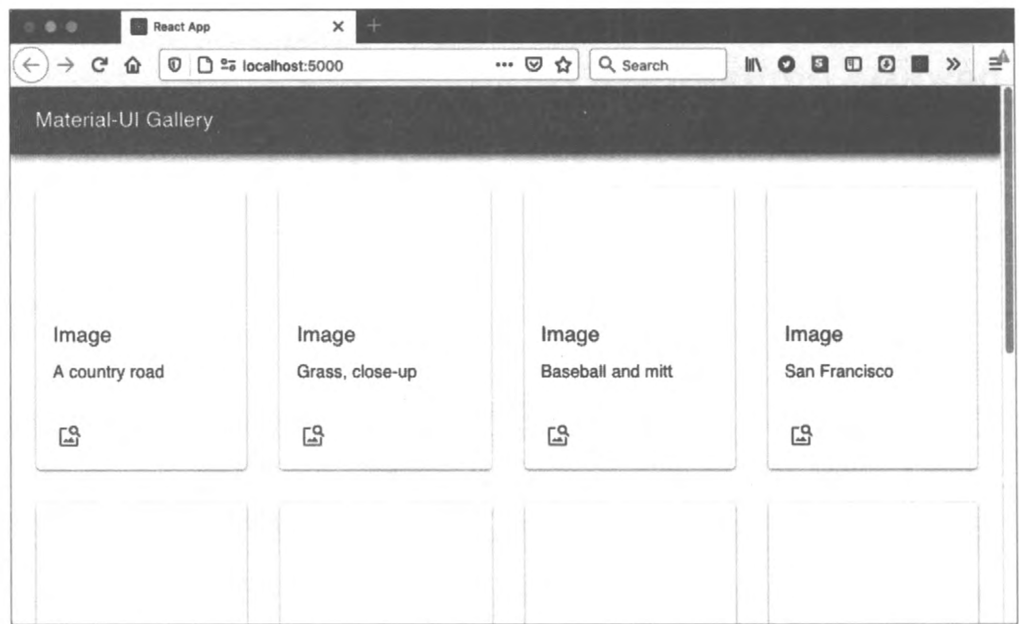


Рис. 11.29. Приложение продолжает работать в автономном режиме, но без изображений

РЕШЕНИЕ

Решим поставленную задачу с помощью пары методов с применением сервис-воркеров, которые сообща будут возвращать локальную замену недоступного изображения из кешированного файла.

Предположим, что нам нужно заменить все недоступные изображения подстановочной картинкой, изображенной на рис. 11.30.

Рис. 11.30. Подстановочная картинка для замены недоступных изображений

Первым делом нам нужно проверить, что файл подстановочного изображения доступен в локальном кеше. Мы добавим этот файл в используемые приложением статические файлы, но мы не можем полагаться на то, что подстановочная картинка будет автоматически сохранена в кеше. При предварительном кешировании сохраняются файлы, загруженные с сервера. Подстановочное изображение не будет требоваться до тех пор, пока присутствует сетевое подключение, поэтому нам придется прибегнуть к методу так называемого разогрева кеша (cache warming) и загрузить изображение в локальный кеш напрямую.

Далее, сразу же после установки сервис-воркера он исполнит определенный код. Для этого нам нужно добавить в него обработчик установки:

```
self.addEventListener('install', (event) => {
  // Сюда вставляется код для кеширования файла изображения
})
```

Мы можем явно открыть локальный кеш `fallback`, а затем добавить в него файл из Сети, как показано в листинге 11.41.

Листинг 11.41. Добавление в кеш файла из Сети

```
self.addEventListener('install', (event) => {
  event.waitUntil(
    caches.open('fallback').then((cache) => {
      cache.add('/comingSoon.png')
    })
  )
})
```

Данный метод позволяет сохранить файлы в локальном кеше при установке приложения. Это полезная возможность при наличии файлов, которые требуются в автономном режиме, но не загружаются приложением сразу же.

Сохранив подстановочную картинку, нам нужно предоставить ее, когда настоящие изображения недоступны. Для этого необходимо добавить в приложение код, который будет выполняться, когда сетевые запросы завершаются ошибкой. Этим кодом может быть *обработчик блока* `catch`, который исполняется при сбое стратегии Workbox, как показано в листинге 11.42.[14]

Листинг 11.42. Исполнение обработчика блока catch

```
setCatchHandler(({ event }) => {
  if (event.request.destination === 'image') {
    return caches.match('/comingSoon.png')
  }
  return Response.error()
})
```

Обработчик блока `catch` принимает в качестве параметра объект неуспешного сетевого запроса. Можно было бы проверить URL-адрес этого запроса, но лучше выяснить его назначение и учесть это при выборе подстановочного объекта для файла. Если назначением является изображение `image`, то запрос был создан вследствие попытки загрузки элемента изображения `img` браузером. В табл. 11.1 приведено несколько других примеров назначений запроса.

[14] Дополнительная информация о библиотеке Workbox предоставляется в других рецептах этой главы.

Таблица 11.1. Примеры назначений запроса

Назначение	Создатель
""	Сетевые запросы JavaScript
"audio"	Загрузка элемента `<audio>`
"document"	Переход на веб-страницу
"embed"	Загрузка элемента `<embed>`
"font"	Загрузка шрифта в таблице стилей
"frame"	Загрузка элемента `<frame>`
"iframe"	Загрузка элемента `<iframe>`
"image"	Загрузка элемента `<img>`, файла /favicon.ico, изображения SVG или изображения таблицы стилей
"object"	Загрузка элемента `<object>`
"script"	Загрузка элемента `<script>`
"serviceworker"	Загрузка сервис-воркера
"sharedworker"	Загрузка общего воркера
"style"	Загрузка таблицы стилей
"video"	Загрузка элемента `<video>`
"worker"	Загрузка воркера

При вызове обработчика блока `catch` он возвращает из кеша изображение comingSoon.png. Чтобы найти этот файл в любом доступом кеше, используется метод `caches.match`.

Теперь, когда у нас есть обработчик блока `catch`, нам нужно определить стратегию Workbox для каждого запроса. При отсутствии такой стратегии сбойный запрос может не активировать этот обработчик. Если задать обработчик по умолчанию, то он будет применять определенную стратегию с каждым запросом, не обрабатываемым каким-либо другим способом:

```
setDefaultHandler(new NetworkOnly())
```

Данная команда обеспечивает перенаправление сервис-воркером всех запросов в Сеть, если только не определен какой-либо более специфичный обработчик.

Каждый из тегов `<img>` страницы генерирует запрос с назначением `image`. Обработчик по умолчанию перенаправляет эти запросы стороннему серверу, что вызовет ошибку, поскольку приложение не имеет доступа к Сети. После этого обработчик блока `catch` возвращает подстановочное изображение для каждого элемента `img`. Результаты этого процесса показаны на рис. 11.31.

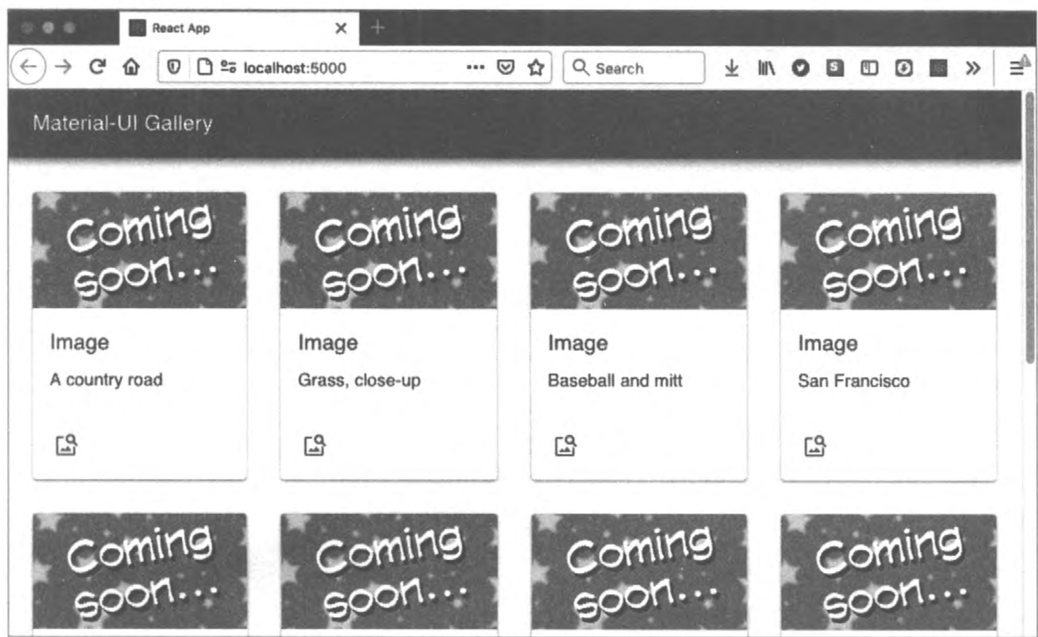


Рис. 11.31. В режиме офлайн все сторонние изображения на странице заменяются подстановочной картинкой

Обсуждение

Описанный метод будет полезным для работы с большими мультимедиафайлами, которые трудно или невозможно сохранить в локальном кеше. Например, в приложении для воспроизведения подкастов недостающий эпизод можно заменить коротким аудиоклипом, информирующим, что данный эпизод будет доступным только при восстановлении подключения к Сети.

Разогрев кеша необходимыми файлами может увеличить время, требуемое для установки сервис-воркера. Поэтому в случае разогрева кеша сравнительно большими файлами также нужно добавить следующий код в соответствующий сервис-воркер:

```
import * as navigationPreload from 'workbox-navigation-preload'
...
navigationPreload.enable()
```

Предварительная загрузка навигационных данных (navigation preload) представляет собой метод оптимизации браузера, заключающийся в исполнении в фоновом режиме сетевых запросов, начавшихся при установке сервис-воркера. Не все браузеры поддерживают этот метод, но при его наличии библиотека `workbox-navigation-preload` использует его.

Исходный код данного рецепта можно загрузить на веб-сайте GitHub по адресу **https://oreil.ly/5nN80**.

Предметный указатель

C

create-react-app 19, 20
CSS-in-JS 164

H

Husky 295

I

ImageMagick 335

J

Jest 49

M

Material Design
◊ карточка 218
◊ спецификация 215
◊ стандарт 71
Material-UI
◊ библиотека 70, 214, 215, 222, 227
◊ библиотека значков 215
◊ темы 217

N

NavLink 62
Next.js 31, 32
npx 20

P

Preact 33, 36
Preact CLI 35

R

Razzle 28, 30

S

SSR 33

W

WCAG 356
Web vitals 416
Webpack 19, 24
Webpacker 39, 41

Y

yarn 20
YubiKey 256

А

Анимация
◊ pulse 165
◊ shake 166
◊ zoomOut 166
Атрибут
◊ allowCredentials 262
◊ aria-haspopup 368
◊ aria-hidden 407
◊ as 224
◊ authenticationSelection 258
◊ autoComplete 297
◊ autoCorrect 296
◊ backgroundColor 383
◊ data-testid 304
◊ excludeCredentials 258
◊ role 363
◊ tabIndex 376, 377
Аттестация 271
◊ объект 259
Аутентификация
◊ в Интернете 256
◊ двухфакторная 256

Б

Библиотека
◊ @material-ui 141
◊ @testing-library/react 48
◊ Animate.css 164
◊ axe-core 384, 386
◊ Axios 192, 253
◊ Bootstrap 223
◊ Day.js 112
◊ Enzyme 48
◊ Focus Lock 406
◊ InfernoJS 37
◊ Jest Dom 301
◊ js-cookie 286
◊ Loadable Components 458
◊ Material-UI 70, 214, 215, 222, 227
▫ установка 215
◊ Moment.js 112
◊ preact-compat 37
◊ Radium 164
◊ React Animations 164
◊ React Bootstrap 223
◊ React Redux 116, 119
◊ React Scripts 19
◊ React Window 227
◊ react-admin 236
◊ react-dom/server 452
◊ react-helmet 27
◊ react-md-editor 158
◊ react-media 53
◊ react-router-dom 51, 53, 54, 60, 66, 69, 75, 76
◊ react-scripts 23, 24
◊ react-transition-group 76
◊ Redux 116, 123, 195
◊ Redux Persist 124, 127
◊ require-text 203
◊ Reselect 128, 130
◊ Selenium 300
◊ Semantic UI 214, 241
◊ Semantic UI React 241
◊ SimpleWebAuthn 266
◊ Sinon 347
◊ svgr 453
◊ Testing Library 300, 301, 304
◊ TweenOne 169
◊ User Event 301, 306
◊ webauthn-json 258
◊ Webpacker 39, 41
◊ Workbox 485
◊ workbox-background-sync 503, 504
◊ workbox-core 466
◊ workbox-navigation-preload 515
◊ криптографическая OpenSSL 268
Блок catch
◊ обработчик 513
Бэкенд, служба
◊ /api/login 251
◊ /api/logout 251

В

Веб-драйвер
◊ chromedriver 327
◊ geckodriver 327

Г

Группа перехода 77

Д

Дерево, встряхивание 441
Диспетчер пакетов
◊ npm 20
◊ yarn 20

З

Заглушка ленивая 438
Загрузка предварительная навигационных данных 515
Задача блокирующая 417
Запрос
◊ на аттестацию 257
◊ на утверждение 262

И

Инструмент
◊ Android Debug Bridge 290
◊ compare 334, 338, 339
◊ производительности Lighthouse 414, 415
Инструменты
◊ набор
 ▫ nwb 37
 ▫ React Developer Tools 419
 ▫ Workbox 465
Интерфейс
◊ Web Speech 410
◊ window.PublicKeyCredential 257
Источник маркера 194

К

Карточка Material Design 218
Кеш, разогрев 512, 515
Класс
◊ galleryGrid 217
◊ galleryItemDescription 217
◊ Modal 369
Клиент
◊ GraphQL 205
◊ Playground 204
Ключ YubiKey 256
Команда
◊ build 22
◊ checkA11y 386
◊ create-razzle-app 29
◊ Cypress 49, 50
 ▫ cy.clock 351, 352, 355
 ▫ cy.contains 318
 ▫ cy.get 318
 ▫ cy.intercept 319, 322
 ▫ cy.network 324, 326
 ▫ cy.random 349
 ▫ cy.tick 352
◊ driver.get 330
◊ eject 23
◊ findElement 330
Компилятор Babel 34
Компонент
◊ Admin 234
◊ AppBar 216
◊ AppoloProvider 206
◊ BrowserRouter 54
◊ Col 224
◊ CssBaseline 215
◊ CSSTransition 77, 80
◊ Dialog 231
◊ Feed 244
◊ FixedSizeList 228, 229
◊ Form 224
◊ Media 53
◊ Modal 369, 406
◊ MyInstaller.js 507
◊ NavLink 62
◊ PeopleContainer 53–55, 57
◊ PeopleList 55
◊ Popper 141
◊ Profiler 413, 422, 427
◊ Prompt 69, 71
◊ Provider 118
◊ ReactFocusLock 406
◊ react-markdown 158
◊ Redirect 56
◊ SecureRoute 84
◊ SecurityContext 84
◊ SecurityProvider 83, 248
◊ Seitch 54
◊ Skip 398, 399
◊ SkipButton 389, 390
◊ StaticRouter 457
◊ TasksContexts 396
◊ Toolbar 216
◊ TransitionGroup 77
◊ Typography 216
◊ withMobileDialog 230

Конечная точка /startRegister 257
Контейнер Suspense 439
Контекст 104, 134
◊ SecurityContext 82
Корневой сертификат 269

М

Маркер
◊ источник 194
◊ отмены 194, 202
Маршрут 477
Маршрутизатор
◊ BrowserRouter 66, 71, 82
◊ MemoryRouter 66
Маршрутизация декларативная 51
Массив renders 425
Метаданные 360
Метод
◊ ByTestId 304
◊ caches.match 514
◊ driver.get 330
◊ driver.wait 333
◊ event.notification.close 498
◊ Event.prompt 508
◊ findAllBy 304
◊ findBy 304
◊ get.Time 355
◊ getAllBy 303
◊ getBy 303
◊ getByText 306
◊ input.sendKeys 331
◊ Math.random 347–349
◊ openWindow 498
◊ queryAllBy 303
◊ queryBy 303
◊ ReactDOM.hydrate 455
◊ ReactDOM.render 455
◊ register 42
◊ setInterval 113
◊ showNotification 496
◊ SSR 33
◊ submit.click 331
◊ time 432
◊ timeEnd 432
◊ toLocaleString 355
◊ waitFor 307
Методы
◊ find... 307
◊ get... 306
◊ query... 306
Модель
◊ DOM 33
◊ виртуальная
  ▫ DOM 448
Модуль
◊ axe DevTools 379, 384
◊ BackgroundSyncPlugin 504
◊ cypres-axe 384
◊ eslint jsx-a11y 372, 373, 378
◊ PathPlugin 171
◊ preset-react 449
◊ workbox-routing 477
◊ workbox-webpack-plugin 470
Модульный тест 300
Мутация 203

Н

Набор инструментов
◊ nwb 37
◊ React Developer Tools 419
◊ Workbox 465
Навигация, средства 359
Настройка privacy.reduceTimerPrecision 433

О

Оболочка совместимости 44
Обработчик
◊ блока catch 513
◊ сервис-воркер onupdatefound 470
Объединитель 129
Объект
◊ Error 134
◊ screen 303
◊ аттестации 259
◊ высказывание 410
◊ деструктурирование 178
◊ сервис-воркер
  ▫ registration 469, 496
  ▫ self.clients 498
Опция --template typescript 23
Ориентиры 371
Очередь endPointQueue1 504
Ошибка CORS 486, 487

П

Пакет
◊ eslint-plugin-react-security 293
◊ инструментов ImageMagick 335

Пакеты
◊ диспетчер
▫ npm 20
▫ yarn 20
◊ сборщик
▫ Webpack 19, 24, 470
Папка build 22
Передающая сторона 258
Переменная
◊ maxRetentionTime 504
◊ NODE_ENV 409
◊ talkToMe 409
◊ среды TZ 354
Побочные эффекты 196
Подкомпонент
◊ Form.Control 224
◊ Form.Group 224
◊ Form.Label 224
◊ Form.Row 224
Показатель
◊ CLS 418
◊ FCP 416, 442
◊ FID 418
◊ LCP 417
◊ SI 417
◊ TBT 417
◊ TTI 417, 442
Правило
◊ no-onchange 374
◊ tabindex 407
Преобразователь 88, 147
◊ данных 87
Приложение одностраничное 19
Программа Eruda 342
Производительность
◊ инструмент Lighthouse 414, 415
Прокси-сервер Selenium Grid 334, 337
Протокол
◊ Chrome DevTools 324
◊ DevTools 325
◊ HTTPS 267

Р

Разделение кода 437
Рассказ 46
Редактор Markdown 157
◊ руководство 161
Роль 363
◊ list 365
◊ listitem 364
◊ menu 367
◊ menuitem 367

С

Сборщик пакетов Webpack 19, 24, 470
Свойство
◊ fullScreen 231
◊ getUserConfirmation 71
◊ navigator.onLine 114
◊ navigator.service Worker.controller 495
◊ query 53
◊ title 369
◊ CSS transition 161
Сервер
◊ API 275
◊ Storybook 335
Сервис-воркер 462
◊ обработчик onupdatefound 470
◊ объект
▫ registration 469, 496
▫ self.clients 498
◊ стратегия 463
Сертификат
◊ SSL самозаверенный 267
◊ корневой 269
Синхронизация фоновая 505
Служба
◊ GTmetrix 418
◊ Sentry.io 134
◊ WebPageTest 418
◊ бэкенд
▫ /api/login 251
▫ /api/logout 251
Событие
◊ beforeinstallprompt 507
◊ focusin 409
◊ notificationclick 498
◊ onbeforeinstallprompt 507, 510
◊ блокирующее 418
Содержимое страницы основное 360
Сообщение SKIP_WAITING 491
Спецификация Material Design 215
Средства навигации 359
Средство
◊ create-react-app 19, 20, 23, 24
◊ eslint 372
◊ ESlint 292
◊ Gatsby 24
◊ Google Analytics 461
◊ Next.js 31, 32

Средство *(прод.)*
◊ Preact 33, 36
◊ Razzle 28, 30
◊ Storybook 45, 48
Ссылки ref 281
Стандарт
◊ Material Design 71
◊ WebAuthn 256, 257, 271
Сторона передающая 258
Стратегия
◊ cache-first 486
◊ Network Only 504
◊ stale-while-revalidate 485, 486
◊ сервис-воркер 463
Схема именования BEM 162
Счетчик состояний 184

Т

Темы Material-UI 217
Тест
◊ Cucumber 300
◊ модульный 300
Технология CSS-in-JS 164
Точка конечная /startRegister 257
Транспилер 23

У

Установка
◊ библиотеки Material-UI: 215
◊ средство eslint 372
◊ шрифта Roboto 215
Утверждение 271
◊ запрос 257, 262
Утилита npx 20

Ф

Файл
◊ CSR 269
◊ cypress/plugins/index.js 384
◊ myprivate.crt 270
◊ myprivate.key 270
◊ serviceWorkerRegistration.js 481
◊ ssr.js 449
Формат Jest 49
Фреймворк
◊ appolo-server 203
◊ Cypress 48, 300, 384
◊ Gatsby 458
◊ Next.js 31
◊ Rails 39
◊ React 19
◊ React Admin 233
◊ Selenium 327
  ▫ установка 335
Функция
◊ applyMiddleware 199
◊ buildGraphQLProvider 235
◊ clientsClaim 466
◊ combineReducers 118
◊ console.table 343
◊ createHanderBoundToURL 477
◊ Date 432
◊ dispatch 93
◊ fetch 502
◊ focusNextElement 400
◊ focusPreviousElement 401
◊ forwardRef 391
◊ get 263
◊ getDescription 409
◊ getUserConfirmation 73, 75
◊ h 36
◊ history.push 57
◊ lazy 437, 438, 457
◊ makeStyles 217
◊ merge 166
◊ navigator.credentials.create 258
◊ navigator.sendBeacon 460
◊ navigator.serviceWorker.controller 470
◊ navigator.serviceWorker.register 469
◊ onAnimationEnd 166
◊ onInstal 482
◊ onRender 424, 429
◊ onUpdate 482, 483, 490
◊ performance.now 426, 433, 434
◊ precacheAndRoute 466
◊ Promise.all 330, 445
◊ Radium.keyframes 166
◊ register 467, 482, 484
◊ registerRoute 485
◊ registerWorker 467
◊ render 302, 303
◊ renderToString 452
◊ reportWebVitals 459
◊ serviceWorkerRegistration.register 482, 490
◊ setTimeout 496
◊ skipWating 491

◊ StaleWhileRevalidate 486
◊ startTask 495
◊ talkToMe 411
◊ terminalLog 386
◊ trace 426, 427
◊ tracker 425
◊ WorkerRegistration.unregister 482
◊ селекторная 87, 128

Хранилище 118
Хук
◊ Husky 295
◊ pre-commit 295
◊ useDispatch 119
◊ useEffect 179
◊ useLocation 77
◊ useMutation 207
◊ useQuery 207
◊ useReduce 91
◊ useStyles 217

Ш

Шаблон
◊ cra-template-pwa 466
◊ стартовый Gatsby 28
Шрифт Roboto, установка 215

Эффекты побочные 196

Об авторах

Дэвид Гриффитс — автор и инструктор. Он занимается разработкой кода в React на профессиональном уровне. Настоящая книга "React. Сборник рецептов" уже шестая на его счету. Его деятельность была связана с разработкой приложений для стартапов, магазинов розничной торговли, производителей транспортных средств и национальных спортивных организаций. Живет и работает в Великобритании.

Дон Гриффитс — автор и инструктор, за спиной которой свыше 20 лет опыта в области разработки программного обеспечения для настольных компьютеров и Интернета. В список ее работ входят книги по созданию приложений для Android, языку программирования Kotlin и по статистике.

Вместе Дэвид и Дон написали несколько книг серии Head First, включая книги *Head First Android Development* и *Head First Kotlin*, а также внесли свой вклад в написание книги *97 Things Every Java Programmer Should Know* ("97 вещей, которые должен знать каждый программист Java"). Они разработали видеокурс Agile Sketchpad как способ обучения фундаментальным принципам и методам таким образом, чтобы слушатели были постоянно активными и вовлеченными. Также они разрабатывают онлайновые инструкции для компании O'Reilly в реальном времени.

Об обложке

На обложке данной книги изображена большая голубая цапля (Ardea herodias). Эти птицы, которых иногда еще называют журавлями, обитают на просторах Северной Америки: размножаются в Канаде, зимуют в Мехико и проводят бóльшую часть года в Соединенных Штатах. Больших голубых цапель часто можно встретить на берегах пресноводных озер и водоемов, рек, а также на окраинах болот, лиманов и прудов. Их также можно увидеть пасущимися на лугах, полях и на других открытых травянистых местностях.

Большие голубые цапли — самые большие цапли в Северной Америке. Их характерная особенность — длинные ноги и шея, а также толстый клюв, который часто сравнивают с кинжалом. На расстоянии они кажутся серо-голубыми, а благодаря их оперению они выглядят так, словно покрыты шерстью. В Южной Флориде и на Карибских островах можно встретить вид цапли полностью белого цвета, называющейся большой белой цаплей, хотя относительно их идет определенный спор, являются ли они отдельным видом. Популяции цапель в восточной части США мигрируют на Карибские острова, в Центральную Америку или на север Южной Америки, в одиночку или стаями. Те, которые живут вдоль тихоокеанского побережья, менее склонны к миграции и могут постоянно проживать в одном месте, забираясь даже далеко на север в юго-восточную Аляску. Касательно своей диеты большие голубые цапли являются оппортунистами и едят почти все, что могут поймать или насадить на свой клюв, включая рыбу, амфибий, пресмыкающихся, небольших млекопитающих и даже других птиц!

В течение периода размножения большие голубые цапли добывают корм на расстоянии нескольких миль от своих гнезд. Колонии из нескольких сотен пар строят гнезда из веток на деревьях, в кустах или на земле вблизи отдаленных болот, островов, прудов или озер. Эти цапли совершают изысканные церемонии ухаживания и брачные танцы, которые могут включать ритуальные поклоны, передачу веток, а также замену друг друга в гнезде. Пары могут оставаться моногамными в течение сезона, но выбирают новых партнеров каждый год. Оба родителя кормят свое потомство срыгиванием. Через два месяца птенцы цапли уже способны летать, а через месяц после первого вылета они покидают гнездо.

В США популяции больших голубых цапель увеличивались с 1966 г., за исключением группы больших белых цапель в южной Флориде, где повышенное содержание ртути в местных водах привело к значительному уменьшению численности. Как и большинство птиц в США, цапли защищены Законом об охране перелетных птиц. Многие животные на обложках книг издательства O'Reilly находятся под угрозой исчезновения, все они важны для нас.

Иллюстрацию к обложке выполнил Карен Монтгомери (Karen Montgomery) на основе черно-белой гравюры Одюбона.

www.bhv.ru

Дронов В. А.

React 17. Разработка веб-приложений на JavaScript

Отдел оптовых поставок:
e-mail: opt@bhv.ru

Простота и скорость разработки

- Компоненты — классы и функции
- Язык JSX
- Веб-формы
- Валидация
- React Router
- Redux
- Разделение кода
- Отладка
- Публикация
- Рендеринг на стороне сервера

Книга посвящена программированию веб-приложений на языке JavaScript с применением популярного веб-фреймворка React. Дается вводный курс, наглядно, по шагам описывающий разработку несложного веб-приложения — списка запланированных дел. Описываются базовые инструменты: создание и настройка React-проекта, написание компонентов, язык JSX, передача данных между компонентами и создание веб-форм. Рассматриваются полезные дополнительные библиотеки: React Router (навигация), Redux, React Redux и Redux Toolkit (централизованное хранилище данных), Formik (быстрая разработка веб-форм), Yup (валидация), React Reveal (анимационные эффекты) и др. Рассказывается о разделении кода, обработке ошибок, средствах отладки, публикации готового веб-приложения и рендеринге на стороне сервера.

Электронный архив на сайте издательства содержит код описанного в книге веб-приложения и другие полезные файлы.

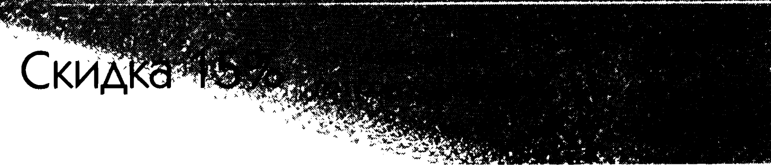
Дронов Владимир Александрович, профессиональный программист, писатель и журналист, работает с компьютерами с 1987 года. Автор более 20 популярных компьютерных книг, в том числе «HTML, JavaScript, PHP и MySQL. Джентльменский набор Web-мастера», «Python 3 и PyQt 5. Разработка приложений» и др. Его статьи публикуются в журналах «Мир ПК» и «ИнтерФейс» (Израиль) и на интернет-порталах IZ City и TheVista.ru.

bhv®

ИНТЕРНЕТ-МАГАЗИН
BHV.RU
КНИГИ, РОБОТЫ,
ЭЛЕКТРОНИКА

Интернет-магазин издательства «БХВ»

- Более 25 лет на российском рынке
- Книги и наборы по электронике и робототехнике по издательским ценам
- Электронные архивы книг и компакт-дисков
- Ответы на вопро



React Сборник рецептов

Фреймворк React поможет вам создать работоспособное приложение всего за несколько минут. Но научиться использовать составляющие React — работа не из легких. Как выполнить валидацию формы? Или реализовать сложное многошаговое действие пользователя, избежав создания запутанного кода? Как протестировать приложение? Приспособить его для многократного использования? Подключить его к бэкенду? Сделать его легким для понимания? Данная книга дает быстрые ответы на эти и другие вопросы.

Многие книги учат, как начать работу с фреймворком React, понять его архитектуру или использовать библиотеки компонентов, но почти нигде не показаны примеры, помогающие решить конкретные задачи. Этот сборник отсортированных по темам и областям применения рецептов содержит примеры кода, которые позволят разработчикам найти решения наиболее распространенных задач при использовании фреймворка React.

Вы научитесь:

- **Создавать одностраничные приложения в React, использующие сложный пользовательский интерфейс**
- **Создавать прогрессивные веб-приложения, которые пользователи смогут устанавить на свои устройства и работать с ними в автономном режиме**
- **Интегрировать разрабатываемые приложения со службами бэкенда, такими как REST и GraphQL**
- **Выполнять автоматическое тестирование для обнаружения проблем с доступностью в разрабатываемом приложении**
- **Обеспечивать безопасность приложений при помощи цифровых отпечатков и ключей безопасности с применением технологии WebAuthn**
- **Исправлять ошибки и избегать распространенных проблем с функциональностью и производительностью**

«Книга проводит читателя через полный жизненный цикл разработки приложений React. Каждый рецепт — это краткий, легко усваиваемый комплекс знаний. Книга обязательна для прочтения всем разработчикам!»

— Сэм Уорнер,
инженер-программист

Дэвид Гриффитс — автор и преподаватель, занимается разработкой кода в React на профессиональном уровне в течение 5 лет. Создавал приложения для стартапов, магазинов розничной торговли, производителей транспортных средств, национальных спортивных команд и крупных поставщиков программного обеспечения.

Дон Гриффитс — автор и преподаватель с более чем 20-летним опытом разработки программного обеспечения для персональных компьютеров и Интернета.

Вместе Дэвид и Дон написали несколько книг, включая Head First Android Development и Head First Kotlin. Также они проводят онлайн-консультации для компании O'Reilly.

ISBN 978-5-9775-6839-5

191036, Санкт-Петербург,
Гончарная ул., 20
Тел.: (812) 717-10-50,
339-54-17, 339-54-28
E-mail: mail@bhv.ru
Internet: www.bhv.ru