PRO ПРОФЕССИОНАЛЬНОЕ ПРОГРАММИРОВАНИЕ

bhv®

React 19

Разработка веб-приложений на JavaScript

- Новинки React 19 и сопутствующих библиотек
- Компоненты
- Работа с бэкендом
- Веб-формы
- Навигация
- Загрузчики и воздействия
- Фетчеры
- Redux
- Анимация
- Работа с метатегами
- Публикация



Владимир Дронов

+ Материалы на www.bhv.ru

Владимир Дронов

React 19
Разработка веб-приложений на JavaScript

Санкт-Петербург
«БХВ-Петербург»
2025

УДК 004.738.5+004.43
ББК 32.973.26-018.2
Д75

Дронов В. А.

Д75 React 19. Разработка веб-приложений на JavaScript. — СПб.: БХВ-Петербург, 2025. — 448 с.: ил. — (Профессиональное программирование)

ISBN 978-5-9775-2036-2

Книга посвящена программированию веб-приложений на языке JavaScript с применением популярного веб-фреймворка React 19. В ней учтены все нововведения, появившиеся в React 19 и новых версиях сопутствующих библиотек. Дается вводный курс, наглядно, по шагам описывающий разработку несложного веб-приложения — списка запланированных дел. Описываются базовые инструменты: создание и настройка React-проекта, написание компонентов, обработка событий, передача данных между компонентами и создание веб-форм, взаимодействие с внешними API и бэкендом. Рассматриваются полезные дополнительные библиотеки: React Router (навигация), Redux, React Redux и Redux Toolkit (централизованное хранилище данных), Formik (быстрая разработка веб-форм), Yup (валидация), React Awesome Reveal (анимационные эффекты) и др. Рассказывается о фетчерах данных, анимации, работе с метатегами, разделении кода, обработке ошибок, средствах отладки и публикации готового веб-приложения.

Электронный архив на сайте издательства содержит полный код описанного в книге веб-приложения.

Для программистов

УДК 004.738.5+004.43
ББК 32.973.26-018.2

Группа подготовки издания:

Руководитель проекта	*Евгений Рыбаков*
Зав. редакцией	*Людмила Гауль*
Редактор	*Григорий Добин*
Компьютерная верстка	*Ольги Сергиенко*
Дизайн серии	*Инны Тачиной*
Оформление обложки	*Зои Канторович*

Подписано в печать 19.11.24.
Формат 70×100 1/16. Печать офсетная. Усл. печ. л. 36,12.
Тираж 1000 экз. Заказ № 11544.
"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 978-5-9775-2036-2

© ООО "БХВ", 2025
© Оформление. ООО "БХВ-Петербург", 2025

Оглавление

https://t.me/it_boooks/2

Предисловие **13**
Веб-сайты третьего поколения. Фронтенды и бэкенды 13
Проблемы с программированием фронтендов 14
Клиентские веб-фреймворки решат все проблемы 14
Веб-фреймворк React — почему именно он? 15
Что содержится в этой книге? 15
Что нового? 16
Необходимые знания и навыки 17
Использованное ПО 17
Типографские соглашения 17

ЧАСТЬ I. СОЗДАНИЕ ПРОСТЕЙШЕГО REACT-ПРИЛОЖЕНИЯ **21**

Глава 1. Список планируемых дел 1.0 ..... **22**
1.1. Создание проекта *todos* 22
Теория 22
Практика 22
1.2. Запуск и остановка проекта 24
1.3. Введение в компоненты. JSX 25
Теория 25
Практика 26
1.4. Вывод компонентов. Пропы и спуск данных 31
Теория 31
Практика 32
1.5. Подъем данных и обработка событий 36
Теория 36
Практика 36
1.6. Состояния компонента. Реактивные данные 38
Теория 38
Практика 38
1.7. Работа с веб-формами 41
1.8. Навигация. Библиотека React Router. Маршрутизатор, маршрут и выпуск 46
Теория 46
Практика 46

1.9. Загрузчики и воздействия React Router. Перенаправление 53
Теория 53
Практика 54
1.10. Параметризованные маршруты 58
Теория 58
Практика 58
1.11. Вывод сообщения об ошибке 404 64

Глава 2. Список планируемых дел 2.0 67
2.1. Создание базы данных Google Firebase и подготовка приложения к работе с ней 67
2.2. Регистрация новых посетителей 77
2.3. Вход и выход 84
2.4. Хранение данных в базе Google Firebase 87
Теория 87
Практика 88
2.5. Разграничение доступа 94
2.6. Валидация и вывод сообщений об ошибках в стиле React 96
Теория 96
Практика 96

ЧАСТЬ II. БАЗОВЫЕ ИНСТРУМЕНТЫ 101

Глава 3. Проект React-приложения 102
3.1. Создание React-проекта 102
3.2. Структура React-проекта 103
3.3. Базовые файлы проекта 106
3.3.1. Веб-страница приложения 106
3.3.2. Стартовый модуль 108
3.3.3. Таблица стилей приложения 109
3.3.3.1. Сброс стилей 109
3.3.3.2. Использование таблиц стилей SASS 110
3.4. Запуск и остановка выполнения проекта 110
3.4.1. Отслеживание кода 112
3.4.2. Использование HTTPS при отладке 112
3.5. Выявление ошибок 113
3.5.1. Ошибки в программном коде 113
3.5.2. Ошибки времени выполнения 114
3.6. Настройка проекта. Переменные окружения 114
3.6.1. Задание переменных окружения. Файлы окружения 115
3.6.2. Использование переменных окружения в программном коде 116
3.6.3. Переменные окружения, поддерживаемые React 116
3.6.4. Создание своих переменных окружения 119

Глава 4. Компоненты 120
4.1. Реализация компонентов 120
4.1.1. Жизненный цикл компонента. Хуки 121
4.1.2. Пропы компонента 122
4.1.3. Рендеринг компонента 124
4.1.3.1. Объектное описание содержимого компонента. Фрагменты 124

4.1.3.2. Разделение кода, выполняющего рендеринг ... 126
4.1.3.3. Предотвращение рендеринга ... 127
4.1.3.4. Рендеринг произвольного содержимого ... 127
4.1.3.4.1. Рендеринг содержимого в одном месте компонента-родителя ... 127
4.1.3.4.2. Рендеринг содержимого в разных местах компонента-родителя. Слоты ... 128
4.1.3.4.3. Инструменты для работы с коллекцией потомков ... 129
4.1.4. Состояния компонента ... 131
4.1.4.1. Состояние компонента: момент первый ... 132
4.1.4.2. Состояние компонента: момент второй ... 133
4.1.4.3. Как состояния связываются с компонентами? ... 134
4.2. Дополнительные инструменты для написания компонентов ... 135
4.2.1. Управление элементами веб-страниц. Рефы ... 136
4.2.1.1. Объектные рефы ... 136
4.2.1.2. Функциональные рефы ... 137
4.2.2. Порталы ... 139
4.2.3. Принудительное обновление компонента ... 141
4.2.4. Генерирование уникальных якорей ... 141
4.3. Повышение производительности компонентов. Мемоизация ... 142
4.3.1. Мемоизация компонентов ... 142
4.3.2. Мемоизация результатов вычислений ... 144

Глава 5. Язык JSX ... 146
5.1. Основы JSX ... 146
5.1.1. Синтаксис JSX ... 146
5.1.1.1. Указание типа выводимого элемента во время выполнения ... 149
5.1.2. Вывод значений ... 150
5.1.3. Условный вывод ... 151
5.1.4. Вывод последовательностей ... 152
5.1.4.1. Ключи ... 152
5.2. Использование в компонентах внедренных элементов ... 153
5.3. Оформление компонентов ... 154
5.3.1. Таблицы стилей компонентов ... 154
5.3.1.1. CSS-модули ... 155
5.3.2. Задание встроенных стилей CSS ... 156

Глава 6. Обработка событий ... 157
6.1. Написание обработчиков событий ... 157
6.2. Привязка обработчиков к событиям ... 158
6.2.1. Передача параметров обработчикам событий ... 159
6.3. События, поддерживаемые React. Класс события React ... 159
6.4. Фазы событий в React ... 161
6.4.1. Обработка событий на фазе погружения ... 162

Глава 7. Взаимодействие между компонентами ... 164
7.1. Передача данных ... 164
7.1.1. Спуск данных ... 164
7.1.1.1. Спуск данных посредством пропов. Простой и сквозной спуск ... 164
7.1.1.2. Спуск данных посредством контекстов ... 166

7.1.2. Подъем данных ... 170
7.1.2.1. Мемоизация функций, передаваемых потомкам ... 171
7.2. Передача функциональности. Рендер-пропы ... 172
7.3. Передача управления ... 174
7.3.1. Перенаправление рефов ... 174
7.3.2. Использование промежуточных обработчиков ... 175

Глава 8. Работа с веб-формами и элементами управления ... 177
8.1. Работа с элементами управления ... 177
8.2. Работа с веб-формами ... 179
8.2.1. Обработка события *submit* веб-формы ... 179
8.2.2. Использование воздействий React ... 180
8.2.2.1. Упрощенные воздействия React ... 180
8.2.2.1.1. Получение сведений о веб-форме ... 182
8.2.2.2. Полнофункциональные воздействия React ... 183
8.3. Валидация заносимых данных ... 185

Глава 9. Работа с внешними API ... 187
9.1. Использование эффектов React ... 187
9.1.1. Эффекты, вызываемые перед рендерингом компонента ... 187
9.1.2. Эффекты, вызываемые после рендеринга компонента ... 190
9.1.3. Эффекты, служащие для взаимодействия с внешним хранилищем данных ... 191
9.2. Поддержка асинхронных операций ... 192
9.2.1. Приостановщик ... 192
9.2.2. Низкоприоритетные обновления компонента ... 194
9.2.2.1. Упрощенные низкоприоритетные обновления ... 195
9.2.2.2. Полнофункциональные низкоприоритетные обновления ... 197
9.2.3. Величины с отложенным изменением ... 198
9.2.4. Оптимистичное обновление компонентов ... 199

Глава 10. Компоненты высшего порядка и собственные хуки ... 202
10.1. Компоненты высшего порядка ... 202
10.1.1. Компонент высшего порядка, пример 1 ... 202
10.1.2. Компонент высшего порядка, пример 2 ... 204
10.2. Написание собственных хуков ... 207

ЧАСТЬ III. ДОПОЛНИТЕЛЬНЫЕ БИБЛИОТЕКИ И РАСШИРЕННЫЕ ИНСТРУМЕНТЫ ... 209

Глава 11. Библиотека React Router: навигация, раздел 1 ... 210
11.1. Основные принципы и понятия React Router ... 210
11.2. Маршруты, списки маршрутов и маршрутизаторы ... 212
11.2.1. Маршрут ... 212
11.2.1.1. Написание шаблонных путей. Запись URL-параметров ... 213
11.2.2. Список маршрутов ... 214
11.2.2.1. Создание списка маршрутов ... 214
11.2.2.2. Две разновидности списка маршрутов ... 214
11.2.2.2.1. Простой список маршрутов ... 214
11.2.2.2.2. Иерархический список маршрутов ... 215

11.2.3. Выпуск ... 218
11.2.4. Маршрутизаторы ... 218
11.2.4.1. Маршрутизатор Browser Router ... 218
11.2.4.2. Маршрутизатор Hash Router ... 219
11.2.5. Провайдер маршрутизации ... 220
11.3. Навигация ... 220
11.3.1. Гиперссылки ... 221
11.3.1.1. Упрощенные гиперссылки ... 221
11.3.1.1.1. Написание путей в гиперссылках ... 222
11.3.1.2. Полнофункциональные гиперссылки ... 223
11.3.2. Перенаправление ... 225
11.4. Получение сведений о текущем интернет-адресе ... 226
11.4.1. Получение значений URL-параметров ... 226
11.4.2. Получение и изменение значений GET-параметров ... 227
11.4.3. Получение текущего местоположения ... 228
11.5. Дополнительные программные инструменты React Router ... 229
11.5.1. Разметочные маршруты ... 229
11.5.2. Обработка события *beforeunload* окна ... 229
11.5.3. Блокировка ухода ... 230
11.5.4. Проверка пути ... 232
11.5.5. Задание маршрутов в виде простых объектов ... 233

Глава 12. Библиотека React Router: навигация, раздел 2 ... 235
12.1. Загрузка и отправка данных ... 235
12.1.1. Загрузка данных. Загрузчики ... 235
12.1.1.1. Написание загрузчиков ... 235
12.1.1.1.1. Перенаправление в загрузчиках ... 236
12.1.1.2. Указание загрузчика в маршруте ... 237
12.1.1.3. Получение результата, возвращенного загрузчиком ... 237
12.1.2. Отправка данных. Воздействия React Router ... 238
12.1.2.1. Написание воздействий React Router ... 238
12.1.2.2. Указание воздействия React Router в маршруте ... 239
12.1.2.3. Инициирование запуска воздействия React Router ... 239
12.1.2.3.1. Запуск воздействия из веб-формы ... 239
12.1.2.3.2. Программный запуск воздействия ... 241
12.1.2.4. Получение результата, возвращенного воздействием React Router ... 243
12.1.3. Фетчеры ... 243
12.1.3.1. Создание фетчеров ... 243
12.1.3.2. Загрузка данных с помощью фетчеров ... 244
12.1.3.3. Отправка данных с помощью фетчера ... 245
12.1.3.3.1. Отправка данных посредством веб-формы фетчера ... 245
12.1.3.3.2. Упрощенный способ отправки данных фетчером из веб-формы ... 246
12.1.3.3.3. Программная отправка данных фетчером ... 246
12.1.3.4. Доступ к произвольному фетчеру ... 247
12.1.3.5. Получение сведений обо всех созданных в приложении фетчерах ... 248
12.2. Поддержка асинхронных операций в React Router ... 248
12.3. Получение сведений о совпавших маршрутах и самом маршрутизаторе ... 251
12.3.1. Получение сведений о совпавших маршрутах ... 251
12.3.2. Получение сведений о самом маршрутизаторе ... 252

12.4. Передача произвольного значения целевому компоненту через маршрут 253
12.5. Сохранение позиции прокрутки 255

Глава 13. Преобразователи данных React 257
13.1. Влияния 257
13.2. Создание преобразователя React 258
13.2.1. Написание преобразователя React 258
13.2.2. Регистрация преобразователя React 259
13.3. Работа с преобразователями React 260

Глава 14. Библиотека Redux: централизованное хранение состояния, раздел 1 262
14.1. Библиотека Redux 262
14.1.1. Влияния 263
14.1.2. Преобразователь Redux 264
14.1.3. Хранилище 265
14.1.4. Селекторы 266
14.1.5. Получение данных из хранилища 267
14.1.6. Отправка влияний 267
14.1.7. Регистрация слушателей 268
14.1.8. Создатели влияний 269
14.1.9. Отправители влияний 270
14.1.10. Специфические случаи применения Redux 271
14.1.10.1. Составные преобразователи 271
14.1.10.2. Замена преобразователя 273
14.1.10.3. Мемоизация селекторов. Библиотека Reselect 273
14.2. Выполнение действий с данными из внешних источников 275
14.2.1. Использование посредников Redux 275
14.2.2. Использование переходников. Библиотека Redux Thunk 280

Глава 15. Библиотека Redux: централизованное хранение состояния, раздел 2 282
15.1. Связывание React-компонентов с хранилищем Redux. Библиотека React Redux 282
15.1.1. Предоставление хранилища компонентам. Провайдер хранилища 282
15.1.2. Доступ к хранилищу посредством функции *connect()* 283
15.1.2.1. Функция *connect()*, параметр № 1: получатель данных 284
15.1.2.2. Функция *connect()*, параметр № 2: брокер влияний 285
15.1.2.3. Функция *connect()*, параметр № 3: составитель пропов 287
15.1.2.4. Функция *connect()*, параметр № 4: настройки 288
15.1.3. Доступ к хранилищу посредством хуков 290
15.1.3.1. Получение данных из хранилища 290
15.1.3.2. Управление данными в хранилище 292
15.1.3.3. Получение доступа к самому хранилищу 292
15.1.3.4. Работа с хранилищем, предоставляемым через пользовательский контекст 293
15.2. Упрощение создания хранилища. Библиотека Redux Toolkit 294
15.2.1. Упрощение формирования создателей влияний 294
15.2.2. Упрощение создания преобразователя 295
15.2.3. Упрощение создания хранилища 299

15.2.4. Упрощение создания преобразователей срезов 301
15.2.4.1. Допустимые и недопустимые типы данных во влияниях 305
15.2.5. Упрощение создания переходников 305
15.2.5.1. Создание переходников 305
15.2.5.2. Создание обработчиков операций 308
15.2.5.3. Запуск переходников, получение результата и обработка ошибок 311
15.2.5.4. Прерывание выполнения операций 312
15.2.6. Создание мемоизированных селекторов 312

Глава 16. Библиотека Formik: создание веб-форм 313
16.1. Быстрое создание веб-форм 313
16.1.1. Компонент *Formik*: построитель веб-форм 313
16.1.2. Компонент *Form*: веб-форма 315
16.1.3. Компонент *Field*: элемент управления 315
16.1.3.1. Вывод флажков, переключателей, списков и полей выбора файлов 316
16.1.4. Обработка занесенных данных 318
16.1.5. Валидация данных 319
16.1.6. Ввод данных в массивы и объекты 322
16.2. Создание более сложных веб-форм 323
16.3. Создание своих элементов управления 328
16.4. Сброс веб-форм 329
16.5. Массивы элементов управления 331

Глава 17. Библиотека Yup: валидация данных 334
17.1. Написание правил валидации. Схемы 334
17.1.1. Правила, применимые ко всем типам 336
17.1.2. Правила валидации строк 337
17.1.3. Правила валидации чисел 339
17.1.4. Правила валидации логических величин 339
17.1.5. Правила валидации временны́х отметок 340
17.1.6. Правила валидации массивов 340
17.1.7. Правила валидации объектов 341
17.1.8. Написание своих дополнительных условий и преобразований 343
17.2. Валидация данных 344
17.2.1. Использование только инструментов Yup 344
17.2.2. Использование Yup совместно с Formik 346

Глава 18. Библиотека ReactFire: интеграция с Google Firebase 347
18.1. Первый уровень: провайдер серверного веб-приложения 347
18.2. Второй уровень: провайдеры базы данных и подсистемы разграничения доступа 348
18.3. Третий уровень: компоненты-потребители 349
18.3.1. Получение сведений о текущем посетителе 349
18.3.2. Извлечение данных из базы 351

Глава 19. Библиотека React Awesome Reveal: анимация 353
19.1. Анимационные эффекты 353
19.1.1. Компоненты, реализующие анимационные эффекты 353
19.1.2. Общие параметры анимационных эффектов 355

19.1.3. Анимирование нескольких элементов (компонентов) 357
19.1.3.1. Анимирование символов текста 358
19.2. Специальные эффекты 358

Глава 20. Прочие полезные инструменты и библиотеки 360
20.1. Библиотека React Window: вывод больших перечней 360
20.1.1. Компонент *FixedSizeList*: вывод списков с пунктами одинакового размера 360
20.1.2. Компонент *VariableSizeList*: вывод списков с пунктами разного размера 364
20.1.3. Компонент *FixedSizeGrid*: вывод таблиц с ячейками одинаковых размеров 365
20.1.4. Компонент *VariableSizeGrid*: вывод таблиц с ячейками разных размеров 369
20.2. Работа с метатегами 371
20.2.1. Указание названия веб-страницы (тег <*title*>) 371
20.2.2. Указание метаданных (тег <*meta*>) 372
20.2.3. Привязка к веб-странице сторонних файлов (тег <*link*>) 372
20.2.4. Задание внутренних таблиц стилей 374
20.2.5. Задание сторонних веб-сценариев 375
20.3. Предварительная загрузка интернет-ресурсов 375

Глава 21. Обработка ошибок 379
21.1. Обработка ошибок средствами JavaScript 379
21.2. Обработка ошибок средствами React Router 380
21.2.1. Базовые средства обработки ошибок 380
21.2.2. Обработка HTTP-ошибок 382
21.2.3. Обработка ошибок в приостановщиках *Await* 383

Глава 22. Разделение кода и загрузка по запросу 384
22.1. Загрузка по запросу средствами JavaScript 384
22.2. Загрузка по запросу средствами React 385
22.3. Загрузка по запросу средствами React Router 386

Глава 23. Расширенные средства отладки 388
23.1. Строгий режим 388
23.2. Настройка отладчика, встроенного в Visual Studio Code 389
23.3. Расширение React Developer Tools 390
23.3.1. Сведения о компонентах 390
23.3.2. Профилировщик 395
23.3.3. Настройка расширения React Developer Tools 399
23.3.4. Упрощение отладки собственных хуков 402
23.4. Расширение Redux DevTools 403
23.4.1. Подготовка проекта к работе с Redux DevTools 403
23.4.2. Работа с Redux DevTools 404
23.4.3. Настройка расширения Redux DevTools 409
23.5. Получение метрик Web Vitals 409
23.6. Встроенный профилировщик React 409

Глава 24. Публикация веб-приложения 412
24.1. Подготовка веб-приложения к публикации 412
24.2. Создание эксплуатационной редакции веб-приложения 413
24.3. Публикация веб-приложения 414

24.3.1. Публикация на мощностях разработчика 414
24.3.1.1. Публикация с помощью Apache HTTP Server 414
24.3.1.2. Публикация с помощью веб-сервера serve 415
24.3.2. Публикация на сторонних мощностях 416
24.4. Выброс React-проекта 416

Заключение 419

Приложение 1. Компоненты-классы 421
П1.1. Введение в компоненты-классы 421
П1.2. Реализация компонентов-классов 422
П1.2.1. Объявление класса компонента 422
П1.2.2. Пропы компонента-класса 422
П1.2.3. Состояние компонента-класса 423
П1.2.4. Написание обработчиков событий в компоненте-классе 425
П1.3. Дополнительные инструменты для написания компонентов-классов 427
П1.3.1. Рефы в компоненте-классе 427
П1.3.2. Получение данных из контекста React 428
П1.3.3. Волшебные методы компонента-класса 429
П1.3.4. Принудительное обновление компонента-класса 433
П1.4. Средства для обработки ошибок в компонентах-классах. Перехватчики 433
П1.5. Повышение производительности компонентов-классов. Лишние обновления и их устранение 435
П1.5.1. Явное подавление лишних обновлений 436
П1.5.2. Создание чистых компонентов 437
П1.6. Отладка компонентов-классов 438

Приложение 2. Устаревший API библиотеки React Router 439
П2.1. Базовые инструменты маршрутизации 439
П2.2. Создание иерархического списка маршрутов средствами устаревшего API 441

Приложение 3. Описание файлового архива 442

Предметный указатель 443

Предисловие

Веб-сайты первого поколения состояли из обычных статических страниц. Они могли лишь показывать посетителям какие-либо тексты, картинки и позволяли загружать файлы. Оставлять на них отзывы и, тем паче, выкладывать на них свои материалы посетители возможности не имели.

Сайты второго поколения представляли собой программы, выполняющиеся на стороне сервера и генерирующие обычные страницы, которые отсылались посетителям. Такие сайты могли получать от посетителей какие-либо данные (например, отзывы или новые статьи для публикации), записывать их в базы данных и позднее выдавать другим посетителям.

Веб-сайты третьего поколения. Фронтенды и бэкенды

Сайты третьего, наиболее актуального сейчас поколения, делают то же самое, только несравнимо быстрее и эффектнее. Они состоят из двух частей:

- *бэкенда* — набора программ, выполняющихся на стороне сервера, обрабатывающих данные, которые хранятся в информационной базе, и пересылающих их на компьютер пользователя в каком-либо компактном формате (обычно JSON);
- *фронтенда* — набора клиентских веб-сценариев, получающих данные от бэкенда и выводящих их на страницы.

Поскольку данные пересылаются в компактном формате, их передача происходит очень быстро. А так как логика вывода данных (фронтенд) отделена от их обработки (бэкенд), чтобы изменить формат вывода данных, достаточно переделать фронтенд, не затрагивая бэкенд. Точно так же, чтобы оптимизировать обработку данных, нужно лишь исправить бэкенд, не касаясь фронтенда. Более того, фронтенд и бэкенд вообще могут писать разные команды разработчиков или даже разные компании.

Сайты третьего поколения в плане функциональных возможностей и быстродействия напоминают обычные настольные приложения. Поэтому их часто называют *веб-приложениями*.

Нередко фронтенд содержит одну-единственную веб-страницу, на которой программно выводится та или иная часть интерфейса приложения (*экран*). Такие веб-приложения носят название *одностраничных*.

В одностраничных приложениях замечательно все! За единственным исключением — их фронтенды чрезвычайно сложно программировать.

Проблемы с программированием фронтендов

Таковых две:

- описать интерфейс приложения (желательно на языке, максимально приближенном к HTML, — так проще);
- обновить интерфейс при изменении подлежащих выводу данных (желательно лишь ту часть интерфейса, в которой выводятся изменившиеся данные, — ради быстродействия).

Можно программно сгенерировать HTML-код, описывающий интерфейс фронтенда, в виде строки, которую присвоить свойству `innerHTML` какого-либо элемента страницы. Однако такой код плохо читается, в нем легко допустить ошибку, и возникают трудности с оптимальным обновлением.

Можно сконструировать интерфейс в виде набора объектов DOM, применив методы `createElement()`, `createTextNode()`, `appendElement()` и т. п. Тогда трудности с оптимальным обновлением в некоторой степени устранятся. Однако получившийся код громоздок и тоже плохо читается.

Есть ли средство устранить эти проблемы? Разумеется, да!

Клиентские веб-фреймворки решат все проблемы

Фреймворк — это программная библиотека, предоставляющая своего рода каркас какой-либо функциональности. *Клиентский веб-фреймворк* — это библиотека, предоставляющая каркас фронтенда.

Хороший клиентский веб-фреймворк:

- позволяет описать интерфейс приложения:
  - либо на языке, очень похожем на HTML;
  - либо на обычном HTML с вкраплениями простого шаблонного макроязыка;
- самостоятельно, без какого бы то ни было участия программиста и наиболее оптимально — обновит интерфейс при изменении выводимых данных.

Хороших веб-фреймворков существует довольно много: «старички» Angular и Vue.js, устаревшие и сошедшие со сцены Backbone и Ember, молодые и полные сил Solid.js, Qwik и Lit, есть еще и другие. Но самый популярный из них, безусловно, React.

Веб-фреймворк React — почему именно он?

Все потому, что React:

- не требует от разработчика изучения никаких дополнительных языков программирования.

  Для описания интерфейса применяется язык JSX, который *очень* похож на HTML, «разбавленный» вставками обычного JavaScript;

- не вносит никаких дополнительных сложностей и ограничений.

  Пишите, как вам удобно, — React покладист;

- быстр.

  Другие фреймворки «тормозят» — React реагирует!

  Что и неудивительно — ведь React изначально был разработан в корпорации Facebook (ныне — Meta) и использовался для вывода страниц одноименной социальной сети. Страниц, чрезвычайно перегруженных сложной разметкой, текстом и графикой;

- имеет разветвленную инфраструктуру поддержки:
  - множество удобных дополнительных библиотек, расширяющих его функциональность;
  - множество сообществ разработчиков, всегда готовых прийти на помощь новичку.

Так что React в настоящее время — самый распространенный в мире веб-фреймворк, написанный на JavaScript. С его применением создано огромное количество веб-приложений.

А количество свободных вакансий React-программистов год от года не уменьшается. Вакансий, надо сказать, весьма высокооплачиваемых...

Вы, уважаемый читатель, можете занять одну из них.
Если прочитаете эту книгу.
Которая как раз посвящена замечательному фреймворку React.

Что содержится в этой книге?

- Для новичков — вводный курс с практическим упражнением...

  ...в котором подробно, доходчиво, по шагам описывается программирование вполне функционального веб-приложения — списка планируемых дел;

- максимально полное описание веб-фреймворка React.

  Рассказ о компонентах и их разработке, языке JSX, обработке событий, передаче данных между компонентами, программировании веб-форм, взаимодействии с бэкендом, а также о создании и настройке React-проектов;

- максимально полное описание дополнительных библиотек, которые обязательно пригодятся React-программистам.

В их число входят: React Router (навигация), Redux, React Redux и Redux Toolkit (централизованное хранение данных), Formik (разработка веб-форм), React Awesome Reveal (анимация) и др.;

- рассказ об инструментах для отладки и выявления ошибок;
- рассмотрение процесса публикации готового веб-приложения;
- и все это — кратко, доходчиво, на многочисленных примерах.

А чего в этой книге точно нет — так это многословных рассуждений ни о чем и прочей «воды».

> ***ФАЙЛОВЫЙ АРХИВ***
>
> Сопровождающий книгу файловый архив содержит программный код учебного приложения списка планируемых дел, рассматриваемого во вводном курсе. Он доступен по ссылке **https://zip.bhv.ru/9785977520362.zip** и со страницы книги на сайте **https://bhv.ru**.

Что нового?

За прошедшие с момента выхода предыдущей книги годы в React появилось много нового. Наиболее значительные и востребованные нововведения в самом фреймворке и библиотеке React Router представлены в табл. П.1.

Таблица П.1. *Наиболее значительные нововведения в React и React Router*

Нововведение	Где описано
Промежуточные обработчики	*Разд. 7.3.2*
Воздействия React	*Разд. 8.2.2*
Эффекты для взаимодействия с внешним хранилищем данных	*Разд. 9.1.3*
Приостановщик	*Разд. 9.2.1*
Низкоприоритетные обновления компонентов	*Разд. 9.2.2*
Величины с отложенным изменением	*Разд. 9.2.3*
Оптимистическое обновление компонентов	*Разд. 9.2.4*
Новый API для описания маршрутов в React Router	*Разд. 11.2*
Блокировка ухода в React Router	*Разд. 11.5.3*
Загрузчики, воздействия и фетчеры React Router	*Разд. 12.1*
Сохранение позиции прокрутки	*Разд. 12.5*
Обработка метатегов	*Разд. 20.2*
Предварительная загрузка интернет-ресурсов	*Разд. 20.3*

Кроме того:

- компоненты-классы объявлены устаревшими и нерекомендованными к применению (за исключением специфических случаев, описанных в *разд. П1.4*);
- библиотека PropTypes больше не используется фреймворком.

Необходимые знания и навыки

- Знание HTML, CSS и современных версий JavaScript;
- навыки веб-разработки (хотя бы базовые);
- знакомство с Node.js (хотя бы поверхностное);
- знание баз данных (хотя бы поверхностное).

Использованное ПО

Таблица П.2. *Версии ПО, использованные автором в процессе написания книги*

Название	Версия	Название	Версия
Node.js	20.14.0, 64-разрядная редакция	create-react-app	5.0.1
React	19.0.0-RC	Bulma	1.0.1
React Router	6.24.0	Google Firebase SDK	10.12.2
Redux	5.0.1	Reselect	5.1.1
Redux Thunk	3.1.0	React Redux	9.1.2
Redux Toolkit	2.2.7	Formik	2.4.6
Yup	1.4.0	ReactFire	4.2.3
React Awesome Reveal	4.2.14	React Window	1.8.10
Visual Studio Code	1.93.20	React Developer Tools	5.3.1
Redux DevTools	3.2.24	serve	14.2.3

Автор работал в среде русской 64-разрядной редакции Microsoft Windows 11 со всеми установленными обновлениями.

Типографские соглашения

В книге будут часто приводиться форматы написания различных языковых конструкций, применяемых в JavaScript- и JSX-коде. В них использованы следующие типографские соглашения:

- в угловые скобки (<>) заключаются наименования различных значений, которые дополнительно выделяются курсивом. В реальный код, разумеется, должны быть подставлены конкретные значения. Например:

  ```
  createReducer(<изначальное состояние>, <строящая функция>)
  ```

  Здесь вместо подстрок *изначальное состояние* и *строящая функция* должны быть подставлены реальные данные — соответственно изначальное состояние для создаваемого хранилища и ссылка на строящую функцию;

- в квадратные скобки ([]) заключаются необязательные фрагменты кода. Например:

```
npx create-react-app[@<версия>] <имя проекта>
```

Здесь параметр *@<версия>* может указываться, но может и отсутствовать (а параметр *имя проекта* обязателен к указанию);

- необязательные *параметры* функций и методов, имеющие *значения по умолчанию*, указываются в формате:

```
<параметр>=<значение по умолчанию>
```

Например:

```
createAction(<тип влияния>[, <подготовитель значения>=undefined])
```

Здесь второй, необязательный, параметр функции `createAction()` имеет значение по умолчанию `undefined`;

- вертикальной чертой (|) разделяются различные варианты языковой конструкции, из которых следует указать лишь какой-то один. Пример:

```
<имя пропа>="<значение пропа>"|{<вычисляемое выражение>}
```

Здесь после знака равенства следует поставить либо *значение пропа* (в двойных кавычках), либо *вычисляемое выражение* (в фигурных скобках);

- слишком длинные, не помещающиеся на одной строке языковые конструкции автор разрывал на несколько строк и в местах разрывов ставил знак ↳. Например:

```
<link rel="preload" imageSrcSet="background2-320w.png, ↳
                                 background2-480w.png 1.5x, ↳
                                 background2-640w.png 2.0x " />
```

Приведенный код разбит на три строки, но должен быть набран в одну. Символы ↳ при этом необходимо удалить;

- троеточием (. . .) помечены фрагменты, пропущенные ради сокращения объема кода. Пример:

```
return (
    <div className="content px-6 py-6">
        . . .
        <main className="content px-6 py-6">
            <TodoList list={todos} />
        </main>
    </div>
);
```

Здесь пропущен фрагмент перед открывающим тегом `<main>`.

Обычно такое можно встретить в исправленных впоследствии частях кода — приведены лишь собственно исправленные выражения, а оставшиеся неизменными пропущены. Также троеточие используется, чтобы показать, в какое

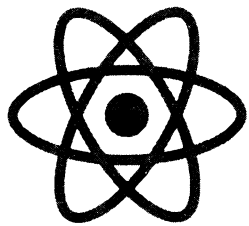
место должен быть вставлен вновь написанный код — в начало исходного фрагмента, в его конец или в середину, между уже присутствующими в нем выражениями;

- полужирным шрифтом выделен вновь добавленный код;
- зачеркнутым шрифтом выделяется код, подлежащий удалению.

Пример:

```
<main className="content px-6 py-6">
    <h1>Todos</h1>
    <TodoList list={todos} />
</main>
```

Здесь заголовок первого уровня (тег `<h1>`) следует удалить, а под ним вставить строку JSX-кода, выводящего компонент `TodoList`.



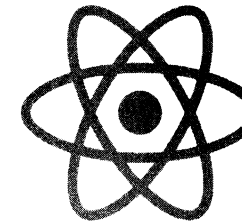
ЧАСТЬ I

Создание простейшего React-приложения

Глава 1. Список планируемых дел 1.0

Глава 2. Список планируемых дел 2.0

ГЛАВА 1



Список планируемых дел 1.0

https://t.me/it_boooks/2

Изучать React лучше на конкретном примере. В этой главе мы начнем программировать простое веб-приложение — список планируемых дел. Оно позволит создавать произвольное количество планируемых дел, выводить их перечень в порядке создания, помечать дела как выполненные и удалять их. Сведения о планируемом деле включат в себя обязательный заголовок, необязательные примечание и графическую иллюстрацию (в качестве которой можно указать, скажем, снимок товара, который следует купить), а также автоматически заносимые дату и время создания.

1.1. Создание проекта *todos*

Теория

Первое, что необходимо сделать, — создать проект нового React-приложения. *Проект* — это совокупность всех файлов, хранящих программный код приложения. Можно сказать, что проект — это и есть разрабатываемое приложение (поэтому программисты часто говорят не «запустить приложение», а «запустить проект»).

Каждый проект хранится в отдельной папке (*папке проекта*), местоположение которой в файловой системе компьютера может быть произвольным. Проект имеет имя, совпадающее с именем этой папки.

Практика

Создадим проект нашего приложения, дав ему имя `todos` (планируемые дела).

1. Установим на компьютер программную платформу Node.js версии 14 или более новой.

   Под управлением платформы Node.js исполняются все инструментальные утилиты, входящие в состав React-проекта. Дистрибутив этой платформы можно найти на странице **https://nodejs.org/en/**.

2. Откроем командную консоль[1] и в ней выполним переход в папку, в которой будет находиться папка создаваемого проекта.

   Например, если планируется поместить новый проект в папке c:\projects\react, следует подать в консоли следующие команды:

   ```
   c:
   cd c:\projects\react
   ```

3. Запустим создание нового проекта `todos`, набрав в консоли команду:

   ```
   npx create-react-app todos
   ```

   Утилита npx, входящая в комплект поставки платформы Node.js, служит для запуска всевозможных программ, написанных под эту платформу, непосредственно из интернет-репозитория Node.js, без необходимости их предварительной установки на локальный диск. При запуске этой утилиты первым параметром указывается имя запускаемой программы, а вторым и последующими — параметры, передаваемые запускаемой программе.

   В нашем случае npx загрузит и запустит инструментальную утилиту create-react-app (ее имя задано первым параметром), посредством которой будет создан новый проект React, и передаст этой утилите при запуске имя создаваемого проекта — `todos` (задано вторым параметром).

   После запуска npx выведет предупреждение о необходимости загрузить из репозитория Node.js программу create-react-app наиболее актуальной версии (показана после имени программы и символа `@` — на момент написания книги актуальной была версия 5.0.1):

   ```
   Need to install the following packages:
   create-react-app@5.0.1
   Ok to proceed? (y)
   ```

4. Ответим положительно на появившееся предупреждение, последовательно нажав клавиши <Y> (от *англ.* yes) и <Enter>.

   После чего утилита npx начнет загрузку заданной программы.

   В процессе загрузки create-react-app и создания React-проекта в консоли будут выводиться всевозможные сведения о ходе процесса. Как только проект будет создан, утилита create-react-app выведет список инструментальных команд, применяемых в процессе отладки проекта (мы рассмотрим их позже), и вернет управление консоли.

   Все загруженные программы сохраняются на локальном диске. При последующем запуске этих программ будут запущены их локальные копии — для повышения производительности.

---

[1] В составе Windows поставляются две выполняющие функции командной консоли утилиты, которые исполняют вводимые с клавиатуры команды: командная строка и Microsoft PowerShell. Можно использовать любую из них.

Всё, новый React-проект `todos` создан. Можете запустить Проводник, открыть в нем папку созданного проекта и посмотреть на ее содержимое.

1.2. Запуск и остановка проекта

Вновь созданный React-проект «пуст», т. е. не содержит никакой полезной логики. Однако он вполне функционален, и его можно запустить на выполнение. Давайте так и сделаем.

1. Откроем консоль и в ней перейдем *непосредственно в папку проекта*.

   Так, если мы в настоящий момент находимся в папке, в которой ранее создали проект, то для перехода в папку проекта наберем команду:

   ```
   cd todos
   ```

2. Запустим проект на выполнение, набрав команду:

   ```
   npm start
   ```

   После этого будет запущен ряд инструментальных программ, входящих в состав проекта и подготавливающих файлы с исходным кодом приложения к передаче веб-обозревателю. Это может отнять довольно много времени (порядка минуты), поэтому следует набраться терпения.

   Как только файлы с кодом приложения будут соответственно подготовлены, запустится отладочный веб-сервер, также входящий в состав проекта и работающий через TCP-порт 3000. Далее автоматически откроется веб-обозреватель, указанный в настройках операционной системы как используемый по умолчанию, который выполнит переход по интернет-адресу **http://localhost:3000/**.

   В результате мы увидим интерфейс пустого React-приложения, включающий вращающийся логотип React, призыв разработчику начать наконец разработку и гиперссылку на домашний сайт фреймворка (рис. 1.1).

   Этот интерфейс формируется компонентом приложения, изначально присутствующим в любом новом React-проекте. В *разд. 1.3* мы познакомимся с компонентами и переделаем компонент приложения под свои нужды.

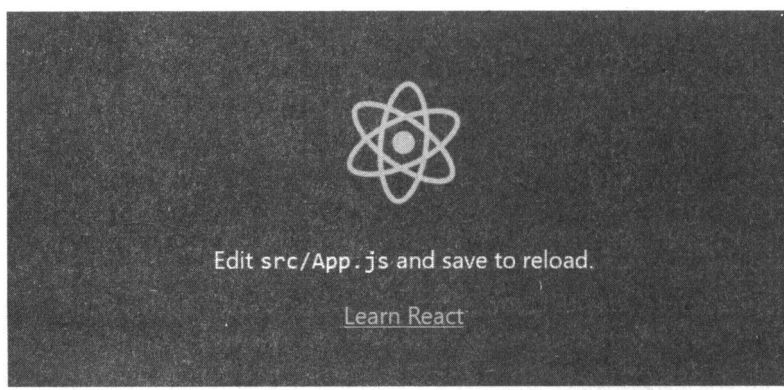


Рис. 1.1. Интерфейс пустого React-приложения

3. Остановим работу проекта, переключившись в экземпляр консоли, из которого ранее запустили его, нажмем комбинацию клавиш <Ctrl>+<C> и подтвердим завершение работы, нажав клавиши <Y> и <Enter>.

> ***На заметку***
>
> Все команды, используемые при работе с проектом, суть сценарии проекта, записанные в файле манифеста проекта *package.json*. Так, введя команду `npm start`, мы тем самым запустим на выполнение сценарий `start`.

> ***Совет***
>
> Если вы пользуетесь текстовым редактором Visual Studio Code, то можете запускать сценарии проекта непосредственно из его среды. Переключите боковую панель редактора на **Проводник**, раскройте секцию **Сценарии NPM**, наведите курсор мыши на нужный сценарий и щелкните на появившейся правее кнопке с изображением стрелки, направленной вправо.
>
> Для завершения работы проекта, запущенного в среде Visual Studio Code, достаточно закрыть экземпляр встроенного терминала, в котором проект был запущен.

1.3. Введение в компоненты. JSX

Теория

Любое веб-приложение, написанное с применением React, содержит в своем составе всего одну веб-страницу (*одностраничное веб-приложение*). В процессе действий посетителя (в частности, навигации) на этой странице программно выводятся различные *экраны* — отдельные части пользовательского интерфейса приложения (их можно рассматривать как аналоги отдельных страниц традиционных многостраничных сайтов). Например, приложение списка планируемых дел в разные моменты времени будет выводить экран с перечнем всех дел, экран со сведениями о деле, выбранном посетителем, экран с веб-формой для добавления нового дела и др.

Далее приведены основные принципы React.

- Интерфейс веб-приложения состоит из взаимосвязанных компонентов.

  Компонент — это независимая часть интерфейса приложения, часто предназначенная для повторного использования. Так, в качестве компонента может быть реализован перечень дел, отдельная позиция этого перечня, кнопка, помечающая дело как выполненное, и т. п.

  Компоненты, составляющие интерфейс React-приложения, можно разделить на три типа:

  - обычные компоненты — реализуют какую-либо часть интерфейса приложения (например, компоненты отдельной позиции перечня дел или кнопки, помечающей дело как выполненное);
  - *компоненты-экраны* — представляют отдельные экраны приложения (например, компонент перечня дел);

- *компонент приложения* — «всеобъемлющий» компонент, который содержит разметку интерфейса приложения и в котором в процессе навигации по приложению выводится тот или иной экран.

Разбиение интерфейса на компоненты дает два преимущества. Во-первых, независимые компоненты проще писать и отлаживать по отдельности. Во-вторых, независимый компонент может быть использован повторно — на другом экране того же приложения или вообще в другом проекте.

- Компонент может содержать любые элементы страницы: абзацы, заголовки, списки, таблицы, изображения, веб-формы, элементы управления, аудио- и видеоролики и др.
- Компонент, помимо обычных элементов страницы, может содержать произвольное количество любых других компонентов. Например, компонент перечня дел может включать набор компонентов-позиций, а компонент-позиция перечня — компонент кнопки, помечающей дело как выполненное.

Компонент, находящийся в составе другого компонента, называется *вложенным*, или *потомком*. Компонент, в который вложен рассматриваемый в текущий момент компонент, является его *родителем*. А два или более компонента, вложенные в один и тот же компонент, носят название *соседей*.

Компонент хранит, обрабатывает и выводит какие-либо данные. Эти данные он может загружать с бэкенда, вырабатывать в результате вычислений или получать от компонента-родителя.

Практика

Сформируем начальный интерфейс нашего веб-приложения, включающий полосу навигации и элементы разметки. Для оформления используем популярный CSS-фреймворк Bulma (**https://bulma.io/**). Он прост в обращении и не требует написания громоздкого HTML-кода.

Разметку следует создавать в компоненте приложения. Этот компонент формируется непосредственно при создании проекта утилитой create-react-app (см. *разд. 1.1*). Его код хранится в модуле src\App.js (а в файле src\App.css хранится таблица стилей этого компонента).

Прежде всего надо установить Bulma и импортировать главную таблицу стилей этого фреймворка. Ее импорт лучше выполнять в модуле src\index.js, который стартует сразу при открытии страницы приложения и запускает это приложение на выполнение (*стартовый модуль*).

1. Откроем консоль, перейдем в папку проекта (если еще не сделали этого) и запустим установку Bulma подачей команды:

```
npm install bulma
```

> ***Совет***
>
> Если вы пользуетесь редактором программного кода Visual Studio Code, вместо отдельной утилиты консоли можете задействовать терминал, встроенный в этот редак-

тор. При выводе нового экземпляра терминала он уже будет открыт в папке проекта (разумеется, если вы предварительно открыли эту папку в самом редакторе).

В коде модуля src\index.js уже присутствует выражение, импортирующее таблицу стилей src\index.css, которая формируется при создании проекта и задает начальное оформление для всего приложения.

2. Откроем стартовый модуль src\index.js в текстовом редакторе и заменим выражение импорта таблицы стилей src\index.css на выражение, импортирующее главную таблицу стилей Bulma из модуля `bulma/css/bulma.css`:

```
. . .
~~import './index.css';~~
import 'bulma/css/bulma.css';
. . .
```

Файл src\index.css после этого можно удалить — больше он нам не понадобится.

Теперь подумаем, в каком виде оформим и где станем хранить сам список планируемых дел, с которым будет работать наше приложение. Оформить список дел удобнее в виде обычного массива JavaScript. А хранить его будем в константе `todos`, которую объявим в отдельном модуле src\todos.js, который сейчас создадим.

Чтобы впоследствии проверить работу компонента перечня дел (который мы напишем уже очень скоро), сразу занесем в создаваемый список пару дел.

Каждое дело, присутствующее в списке, будет представлять собой простой объект (созданный на основе класса `Object` с применением объектной нотации — фигурных скобок) со следующими свойствами:

- `title` — заголовок дела (описывает, что, собственно, нужно сделать);
- `desc` — необязательное примечание к делу;
- `image` — графическая иллюстрация в виде data URL или пустая строка, если иллюстрация отсутствует;
- `done` — если `false`, дело еще не выполнено, если `true` — уже выполнено;
- `createdAt` — автоматически заносимые дата и время создания дела, представленные в виде строки;
- `key` — уникальный идентификатор дела (пригодится в дальнейшем). Будет представлять собой временну́ю отметку создания, оформленную в виде количества миллисекунд, прошедших с полуночи 1 января 1970 года, и также будет записываться автоматически.

Для простоты пока не будем указывать у создаваемых дел примечания и графические иллюстрации.

3. Создадим модуль src\todos.js, откроем его в текстовом редакторе и запишем в него код из листинга 1.1.

Листинг 1.1. Модуль списка планируемых дел (src\todos.js)

```
const date1 = new Date(2024, 6, 18, 10, 24);
const date2 = new Date(2024, 6, 19, 14, 47);

const todos = [
    {
        title: 'Изучить React',
        desc: 'Да поскорее!',
        image: '',
        done: true,
        createdAt: date1.toLocaleString(),
        key: date1.getTime()
    },
    {
        title: 'Написать первое React-приложение',
        desc: 'Список запланированных дел',
        image: '',
        done: false,
        createdAt: date2.toLocaleString(),
        key: date2.getTime()
    }
];
export default todos;
```

Первое дело мы сразу пометили как выполненное — это позволит нам в дальнейшем протестировать вывод выполненных дел.

Теперь можно заняться компонентом приложения. Он будет выводить полосу навигации и разметку.

4. Откроем модуль src\App.js, хранящий код компонента приложения `App`, в текстовом редакторе и полностью заменим его код на представленный в листинге 1.2.

Листинг 1.2. Компонент приложения `App` (модуль src\App.js)

```
import todos from './todos.js'; // ..................................  1

export default function App() { // ..................................  2
    return ( // .....................................................  3
        <div className="container"> // ..............................  5
            <nav className="navbar is-light"> // .....................  6
                <div className="navbar-brand"> // ....................  7
                    <span className="navbar-item is-uppercase"> // ...  8
                        Todos
                    </span>
                </div>
            </nav>
```

```
            <main className="content px-6 py-6"> // ..................  9
                <h1>Todos</h1>
            </main>
        </div>
    ); // ..........................................................  4
}
```

Поскольку код компонента `App` полностью переписан, таблица стилей src\App.css, импортировавшаяся «старым» компонентом и задающая для него оформление, больше не нужна. Также не нужен файл src\logo.svg, содержащий графический логотип, который выводился «старым» компонентом. Оба этих файла можно удалить.

В коде компонента приложения импортируем константу `todos`, объявленную в ранее созданном модуле src\todos.js и содержащую массив дел (поз. 1 в листинге 1.2). Этот массив мы позже передадим компоненту перечня дел.

Компонент React представляет собой обычную функцию JavaScript (поз. 2). Эта функция может выполнять любые вычисления, необходимые для вывода компонента, а также принимать единственный параметр (о котором мы скоро поговорим).

Но главная задача любой функции, реализующей компонент, — вернуть особый объект, описывающий содержимое компонента, выводимое на экран: все имеющиеся в нем абзацы, заголовки, списки, таблицы, элементы управления и пр. (поз. 3).

Писать JavaScript-код, формирующий подобного рода объекты, очень трудоемко. Поэтому разработчики React создали для своих нужд язык *JSX* (JavaScript XML). Его синтаксис очень похож на таковой у языков HTML и XML за двумя исключениями:

- все теги в JSX являются парными. Например, для вставки графического изображения следует использовать парный тег:

  ```
  <img src=". . ."></img>
  ```

  или его сокращенную запись:

  ```
  <img src=". . ." />
  ```

- для указания параметров у тегов в JSX используются не атрибуты HTML, а соответствующие им свойства классов. Так, для указания стилевых классов должен быть использован не атрибут тега `class`, а свойство `className`.

Фрагменты обычного JavaScript-кода, присутствующие в JSX-коде, следует заключать в фигурные скобки.

Поскольку мы для удобства разбили JSX-код на несколько строк, то взяли его в круглые скобки (поз. 3 и 4). Если этого не сделать, возникнет синтаксическая ошибка.

Какие элементы страницы содержит компонент приложения `App`?

Прежде всего, это *корневой тег*, заключающий в себе все содержимое компонента (поз. 5) — таково требование React. В качестве корневого тега мы применили обычный блок (`<div>`).

У корневого блока мы указали стилевой класс `container`. Он привязывает к блоку особый стиль, содержащийся в CSS-фреймворке Bulma и превращающий обычный элемент страницы в контейнер. *Контейнером* в терминологии Bulma называется элемент, размеры которого адаптируются к размерам окна веб-обозревателя, что позволяет создавать страницы, одинаково хорошо выводящиеся на экранах разных размеров, в том числе экранах мобильных устройств.

В корневом теге содержится ряд следующих тегов:

- `<nav>` (поз. 6) — со стилевыми классами `navbar` (создает полосу навигации в стиле Bulma) и `is-light` (задает светло-серый цвет фона) — сама полоса навигации, пока практически «пустая»;
  - блок (поз. 7) — со стилевым классом `navbar-brand` — *обозначение бренда*, выводящееся в левой части полосы навигации и обычно представляющее собой название приложения.

    В этом блоке находится тег `<span>` (поз. 8) со стилевыми классами `navbar-item` (собственно обозначение бренда) и `is-uppercase` (приведение текста к верхнему регистру), в котором и выводится наш бренд — слово «Todos».

  Меню навигации (часть полосы навигации, в которой, собственно, выводятся пункты, предназначенные для перехода на различные экраны приложения) мы создадим потом;
- `<main>` (поз. 9) — со стилевыми классами `content` (оформление содержимого страницы в стиле Bulma), `px-6` (большие внутренние просветы слева и справа) и `py-6` (большие внутренние просветы сверху и снизу) — контейнер для вывода содержимого экранов. Изначально в нем присутствует заголовок «Todos», позже мы заменим его на компонент перечня дел.

Мы только что написали наш первый компонент React! И можем прямо сейчас проверить его в действии. Сохраним все исправленные файлы и запустим проект (как это сделать, было рассказано в *разд. 1.2*). В консоли появится предупреждение о том, что константа `todos`, импортированная в первом выражении модуля src\App.js, нигде не используется, — проигнорируем его. Посмотрим, правильно ли отображается компонент (рис. 1.2).

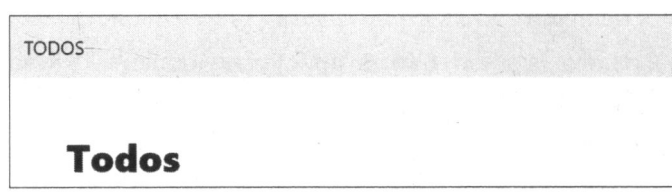


Рис. 1.2. Полоса навигации и тестовый заголовок

Пока не будем завершать работу проекта. Благодаря этому мы познакомимся с одной любопытной и полезной особенностью React, которая очень упростит нам работу.

> ***Совет***
>
> Если вы пользуетесь редактором Visual Studio Code, установите расширение Babel JavaScript. Оно выполняет синтаксическую подсветку специфических конструкций языка JSX, равно как и всех нововведений, появившихся в последних версиях JavaScript, благодаря чему повышается удобство работы.

> ***За кулисами***
>
> Веб-обозреватели не поддерживают JSX, поэтому JSX-код перед исполнением преобразуется (транслируется) в обычный JavaScript. Это выполняет популярная программа-*транслятор Babel*, входящая в состав проекта React. Помимо того, она преобразует код, написанный с использованием последних нововведений в JavaScript, которые поддерживаются только самыми современными веб-обозревателями, в более «старый» и более совместимый.
>
> Другая инструментальная программа из состава React-проекта — *сборщик Webpack* — собирает программный код из всех файлов в один большой массив — *связку*, или *бандл*[1]. Это делается для ускорения загрузки, поскольку один большой массив кода загружается значительно быстрее, чем несколько десятков мелких файлов.

Следующий шаг — программирование компонента, выводящего перечень дел. Мы поместим его в компонент приложения, сделав потомком последнего. В результате компонент приложения автоматически выведет компонент перечня, предварительно передав ему массив со списком дел.

> ***Полезно знать***
>
> React также позволяет реализовывать компоненты в виде классов. Однако компоненты-классы к настоящему моменту объявлены устаревшими и не рекомендуются к применению. О них можно прочитать в *приложении 1*.

1.4. Вывод компонентов. Пропы и спуск данных

Теория

В *разд. 1.3* говорилось, что компоненты могут включать в состав своего содержимого, наряду с обычными HTML-тегами, другие компоненты. Более того, сами разработчики фреймворка React настоятельно рекомендуют по возможности дробить содержимое сложных компонентов на более простые компоненты — для упрощения программирования и сопровождения.

Компонент-родитель при выводе компонента-потомка может передать ему произвольные данные — *пропы*. Каждый проп способен хранить одно значение и должен иметь уникальное имя, по которому компонент-потомок сможет обратиться к нему, чтобы извлечь переданное значение. Передача данных подобного рода — «вниз», или «вглубь» компонента — называется *спуском*.

---

[1] От *англ.* bundle — связка.

Практика

Напишем компонент `TodoList`, выводящий перечень планируемых дел в виде таблицы. Он будет содержать три столбца: заголовок очередного дела и кнопки **Выполнено** и **Удалить**. У выполненных дел заголовок будет выводиться зачеркнутым, а кнопка **Выполнено** станет недоступна.

Массив со списком дел будем передавать в компонент `TodoList` посредством пропа, которому дадим имя `list`.

Сохраним код нового компонента в модуле src\TodoList.js.

1. Создадим модуль src\TodoList.js и запишем в него код из листинга 1.3.

Листинг 1.3. Компонент перечня дел `TodoList` (модуль src\TodoList.js)

```
export default function TodoList(props) { // ..........................  1
    return (
        <section> // ..................................................  2
            <h1>Дела</h1>
            <table className="table is-hoverable is-fullwidth"> // ......  3
                <tbody> // ..............................................  4
                    {props.list.map(item => ( // .......................  5
                        <tr key={item.key}> // .........................  6
                            <td> // .....................................  7
                                {item.done && <del>{item.title}</del>}
                                {!item.done && item.title}
                            </td>
                            <td>
                                <button // ..............................  8
                                    className="button is-success"
                                    title="Выполнено"
                                    disabled={item.done} // ..............  10
                                >
                                    &#9745; // ..........................  11
                                </button>
                            </td>
                            <td>
                                <button // ..............................  9
                                    className="button is-danger"
                                    title="Удалить"
                                >
                                    &#9746; // ..........................  12
                                </button>
                            </td>
                        </tr>
                    ))}
                </tbody>
```

```
            </table>
        </section>
    );
}
```

Функция, реализующая компонент, в качестве единственного параметра может принимать простой объект, хранящий все переданные этому компоненту пропы (поз. 1 в листинге 1.3). Традиционно этому параметру дают имя `props`.

В качестве корневого в компоненте можно использовать любой тег. Так, мы использовали HTML-тег `<section>` (поз. 2).

В теге `<table>`, выводящем таблицу с перечнем дел (поз. 3), мы указали стилевые классы, задающие оформление в стиле Bulma: `table` (основное оформление таблицы), `is-hoverable` (подсветка строк при наведении на них мыши) и `is-fullwidth` (растягивание таблицы на всю ширину родителя). В тег `<table>` вставили тег `<tbody>` (поз. 4) — иначе таблица не получит заданное оформление.

Массив дел был передан компоненту `TodoList` через проп `list`. Следовательно, извлечь этот массив можно из свойства `list` объекта, полученного с единственным параметром `props` функцией, которая реализует компонент (поз. 5).

Чтобы перебрать все дела в полученном массиве, мы использовали метод `map()`, поддерживаемый классом массива `Array`. В стрелочной функции, переданной этому методу, выводим строку с ячейками, содержащими краткое описание очередного дела (поз. 5).

Вызов метода `map()` представляет собой выражение, написанное на обычном JavaScript. Правила языка JSX предписывают заключать JavaScript-код в фигурные скобки (см. *разд. 1.3*). Что мы и сделали.

У тега `<tr>`, выводящего строку таблицы, мы указали параметр `key` и дали ему в качестве значения уникальный идентификатор очередного дела (поз. 6), тем самым связав строку таблицы и дело, которое в ней выводится. Благодаря этому React каждый раз при обновлении компонента:

- если дело, связанное с какой-либо строкой, отсутствует в массиве, — удалит эту строку;
- если дело, связанное со строкой, присутствует в массиве, — обновит существующую строку (что отнимет меньше времени, нежели удаление строки и создание ее заново), а если содержимое дела не изменилось, даже не будет ее изменять;
- для вывода вновь добавленных дел — создаст для их вывода новые строки.

Все это обеспечивает максимально быстрый вывод компонентов на экран.

Выражение языка JavaScript, извлекающее значение свойства `key` объекта дела, мы заключили в фигурные скобки, как предписывают правила JSX.

В первой ячейке таблицы (поз. 7) заголовок дела следует вывести зачеркнутым (для чего можно использовать HTML-тег `<del>`), если дело выполнено, и не зачеркнутым — в противном случае. Для этого мы использовали любопытную

особенность обработки оператора логического И (`&&`). Если его первый операнд равен `true`, этот оператор вернет значение второго операнда без каких бы то ни было преобразований типов, и это значение будет выведено на страницу. Если же первый операнд равен `false`, оператор вернет `false`, в результате чего на страницу ничего выведено не будет.

Выражения, выводящие заголовок дела, мы также заключили в фигурные скобки.

У тегов `<button>`, создающих кнопки (поз. 8 и 9), указаны стилевые классы, задающие оформление в стиле Bulma: `button` (основное оформление кнопки), `is-success` (зеленый цвет) и `is-danger` (красный цвет).

Кнопка, помечающая дело как выполненное, должна быть доступна лишь в том случае, если дело не является выполненным (свойство `done` хранит значение `false`). Мы добились этого, записав в теге `<button>`, выводящем эту кнопку, свойство `disabled` и присвоив ему значение свойства `done` дела (поз. 10). И не забыли заключить выражение, обращающееся к этому свойству, в фигурные скобки.

На кнопке, помечающей дело как выполненное, будет выведен символ галочки, имеющий Unicode-код 9745 (поз. 11), а на кнопке удаления дела — символ крестика с кодом 9746 (поз. 12).

Теперь запишем в компонент `App` код, который выведет компонент `TodoList` внутри тега `<main>` вместо заголовка «Todos».

2. Вставим в код компонента `App` (модуль **src\App.js**) выражение, выводящее компонент `TodoList`, а также выражение, импортирующее его из модуля **src\TodoList.js**:

```
import TodoList from './TodoList.js';
. . .
export default function App() {
    return (
        <div className="container">
            . . .
            <main className="content px-6 py-6">
                <h1>Todos</h1>
                <TodoList list={todos} />
            </main>
        </div>
    );
  }
}
```

Чтобы вывести компонент внутри другого компонента, в нужное место JSX-кода, формирующего содержимое компонента-родителя, следует вставить тег, в качестве имени которого указывается имя выводимого компонента-потомка (*тег компонента*).

Для вывода компонента `TodoList` мы использовали тег компонента:

```
<TodoList . . . />
```

(не забываем, что в JSX все теги — парные).

Чтобы передать массив дел, хранящийся в константе `todos` из модуля src\todos.js, компоненту-потомку `TodoList` в пропе с именем `list`, мы запишем этот проп непосредственно в теге компонента `TodoList`, подобно обычному свойству.

Значение пропа `list` фактически получается в результате выполнения выражения JavaScript, которое извлекает массив дел из константы `todos`. Поэтому мы заключили обращение к этой константе в фигурные скобки (а если бы не сделали этого, проп в качестве значения получил бы совершенно бесполезную строку `'todos'`).

Сохраним оба файла — исправленный src\App.js и вновь созданный src\TodoList.js. Инструментальные средства React-проекта выполнят повторную трансляцию этих файлов (сообщив об этом в консоли, в которой был запущен проект) и отправят их веб-обозревателю, заменив ранее загруженные им предыдущие редакции этих же файлов. В результате запущенное приложение автоматически обновится. И на экране появится перечень с двумя делами (рис. 1.3).

Внимание!

К сожалению, автоматическое обновление запущенного приложения иногда не срабатывает. В таком случае (разумеется, если в коде нет явных ошибок) попробуйте обновить открытое в веб-обозревателе приложение, нажав клавишу <F5>. Если же и это не помогает, завершите работу проекта и запустите его снова.

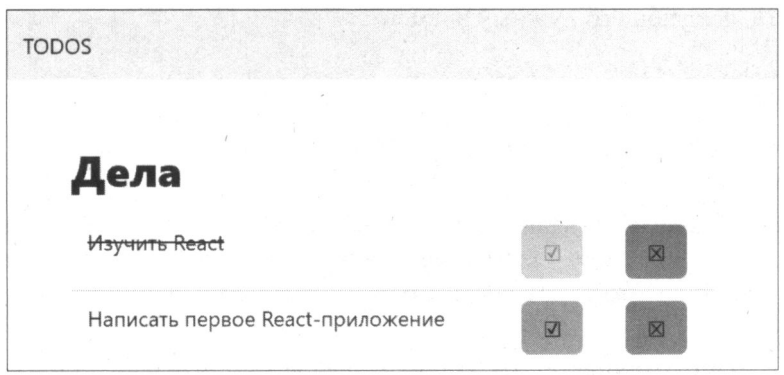


Рис. 1.3. Перечень дел

Совет

В процессе разработки обычно имеет смысл не останавливать работу запущенного проекта, поскольку его повторный запуск может отнять много времени. Останавливать проект следует лишь перед значительными переделками.

За кулисами

Страница приложения *public\index.html*, которая открывается в веб-обозревателе при запуске проекта, содержит блок (тег `<div>`) с якорем `root` (*якорь* — уникальный идентификатор элемента страницы, задаваемый в атрибуте тега `id`). Стартовый модуль *src\index.js*, выполняющийся после открытия страницы, содержит код, который выводит в этом блоке компонент приложения `App`.

Мы только что написали первый экран нашего приложения. Позже, реализуя маршрутизацию, мы добавим еще два экрана.

Но сначала заставим работать кнопки **Выполнено** и **Удалить**.

1.5. Подъем данных и обработка событий

Теория

Спуск данных (т. е. их передача от компонента-родителя компоненту-потомку) реализуется путем передачи данных через пропы. Однако *подъем данных* — их передачу в обратном направлении, от потомка родителю, — выполнить напрямую не получится.

Для этого используется обходной маневр. Компонент-родитель передает потомку через какой-либо из пропов вложенную функцию, объявленную в родителе. Когда потомок намеревается отправить данные родителю, он вызывает эту функцию и передает ей с параметрами отправляемые значения. Вызванная функция принимает отправленные значения и либо выводит их на страницу, либо отправляет в обработку.

React значительно упрощает обработку событий в элементах страницы, составляющих содержимое компонента. Обработчик — вложенная функция, объявленная в компоненте, — привязывается к событию путем присваивания соответствующему свойству тега, создающего нужный элемент.

Практика

Сделаем так, чтобы по нажатию кнопки **Выполнено** соответствующее дело помечалось как выполненное.

Сначала необходимо объявить в компоненте приложения `App` вложенную функцию `setDone(<идентификатор>)`, помечающую дело с указанным *идентификатором* как выполненное. Это будет достигаться занесением в свойство `done` дела значения `true`.

1. Добавим в код компонента `App` (модуль src\App.js) объявление вложенной функции `setDone()`:

```
export default function App() {
    const setDone = key => {
        const deed = todos.find(current => current.key === key);
        if (deed)
            deed.done = true;
    };
    . . .
}
```

Функцию `setDone()` мы реализовали в виде одноименной константы, которой присвоили стрелочную функцию. Поскольку стрелочные функции несколько «легче» обычных, так мы сэкономим немного системных ресурсов.

В теле этой функции используем метод `find()` класса `Array` для поиска в массиве дела, у которого идентификатор совпадает с полученным в качестве параметра. Если дело с заданным идентификатором найдено, заносим в его свойство `done` значение `true`.

Теперь нужно передать ссылку на функцию `setDone()` компоненту `TodoList` через проп, которому также можно дать имя `setDone`.

2. Добавим в код компонента `App` необходимый код:

```
export default function App() {
    . . .
    return (
        <div className="container">
            . . .
            <main className="content px-6 py-6">
                <TodoList list={todos} />
                <TodoList list={todos} setDone={setDone} />
            </main>
        </div>
    );
}
```

Поскольку пропу `setDone` фактически передается значение выражения JavaScript, заключим это выражение в фигурные скобки.

Осталось указать в компоненте перечня дел в качестве обработчика щелчка на кнопке **Выполнено** функцию, полученную с пропом `setDone`.

3. Добавим в компонент `TodoList` (модуль src\TodoList.js) требуемый код:

```
export default function TodoList(props) {
    return (
        . . .
                                <button
                                    className="button is-success"
                                    title="Выполнено"
                                    disabled={item.done}
                                  onClick={() => {props.setDone(item.key)}}
                                >
                                    &#9745;
                                </button>
        . . .
    );
}
```

Привязка функции-обработчика к событию выполняется присваиванием ссылки на эту функцию свойству элемента страницы, имеющему имя формата on<*имя события*>, где в *имени события* каждое слово начинается с заглавной буквы. Так, для привязки обработчика к событию `click` следует использовать свойство

`onClick` (как в нашем случае), а для привязки обработчика к событию `mousemove` — свойство `onMouseMove`.

В качестве обработчика события мы использовали стрелочную функцию, которая вызывает функцию, полученную с пропом `setDone`, передавая ей идентификатор текущего дела.

Сохраним оба файла и попробуем пометить второе дело как выполненное, щелкнув на соответствующей кнопке. В результате... ничего не изменится. Дело в том, что данные, хранящиеся в обычных переменных и константах (наподобие `todos`), не являются реактивными и соответственно не приводят к автоматическому обновлению компонента.

1.6. Состояния компонента. Реактивные данные

Теория

Реактивные данные — это данные, изменение которых приводит к автоматическому обновлению компонента. Типичный пример реактивных данных — пропы, получаемые компонентом извне. При изменении пропов компонент обновляется.

Также к реактивным данным относятся *состояния* компонента — значения, вырабатываемые самим компонентом в процессе работы и хранящиеся в самом компоненте. Как правило, в виде состояния оформляется значение, либо непосредственно выводящееся на экран, либо управляющее выводом различных частей компонента.

Так, в качестве состояния можно оформить массив дел. Тогда при каждом изменении этого массива (добавлении нового дела, пометке дела как выполненного или удаления дела) компонент будет «перерисован», и посетитель увидит на экране обновленный перечень дел.

Практика

Сделаем так, чтобы при пометке дела как выполненного компонент перечня дел автоматически обновлялся. После чего реализуем удаление дел.

Сначала превратим массив дел в состояние компонента `App`.

1. Исправим код компонента `App` следующим образом:

```
import { useState } from 'react';
. . .
import todos from './todos.js';
import initialTodos from './todos.js'; // ..............................  1

export default function App() {
    const [todos, setTodos] = useState(initialTodos); // ..............  2
    . . .
}
```

Для упрощения программирования дадим создаваемому состоянию имя `todos`. Чтобы избежать конфликта имен с константой, импортированной из модуля src\todos.js и хранящей изначальный массив дел, мы импортируем эту константу под именем `initialTodos` (поз. 1 в представленном коде).

Состояние создается вызовом функции `useState()`, объявленной в модуле `react` (поз. 2). В качестве параметра этой функции передается значение, которое будет занесено в создаваемое состояние изначально (у нас это исходный массив дел из константы `initialTodos`).

Состояние создается при первоначальном выводе компонента на экран (его *монтировании*), и в него заносится значение, указанное в вызове функции `useState()`. Созданное состояние будет существовать все время, пока компонент, которому оно принадлежит, присутствует на экране. После удаления компонента с экрана (*размонтировании*) принадлежащее ему состояние также будет удалено.

Функция `useState()` возвращает массив из двух элементов:

- текущего значения состояния;
- *сеттера состояния* — функции, изменяющей значение состояния (а изменение состояния, как говорилось ранее, вызовет обновление компонента). Единственным параметром этой функции передается новое значение состояния.

  Следует учесть, что React перед изменением состояния сравнивает текущее значение состояния и новое значение, которое требуется в него занести. Если значения равны, изменения состояния (и соответственно обновления компонента) не произойдет.

  Поэтому, если значением состояния является массив или объект, необходимо создать его копию, внести требуемые изменения в нее, после чего эту копию передать сеттеру. Только после этого состояние будет изменено, и компонент обновится.

Как правило, массив, возвращаемый функцией `useState()`, подвергается деструктурированию. Так поступили и мы, сохранив первый элемент массива (текущее значение состояния) в константе `todos`, а второй элемент (сеттер) — в константе `setTodos`.

`useState()` относится к разряду функций, позволяющих расширить функциональность компонентов, или *хуков*. Имена всех хуков начинаются с префикса `use`.

Создав состояние, хранящее массив дел, мы сможем реализовать пометку дела как выполненного с автоматическим обновлением перечня дел.

2. Внесем в код компонента `App` следующие правки:

```
export default function App() {
    . . .
```

```
    const setDone = key => {
        const newTodos = [...todos]; // ..............................  1
        ~~const deed = todos.find(current => current.key === key);~~
        const deed = newTodos.find(current => current.key === key); // ..  2
        if (deed)
            deed.done = true; // ....................................  3
        setTodos(newTodos); // ..................................  4
    };
    . . .
}
```

Сначала мы создаем копию массива дел на основе оригинала массива, хранящегося в состоянии, который мы получили ранее вызовом хука `useState()` и сохранили в константе `todos` (поз.1 в представленном коде). Далее в полученной копии массива ищем дело с заданным идентификатором (поз. 2) и помечаем найденное дело как выполненное (поз. 3). Напоследок заносим исправленную копию массива дел в состояние, воспользовавшись сеттером, полученным ранее тем же вызовом хука `useState()` и хранящимся в константе `setTodos` (поз. 4). Изменение состояния вызовет автоматическое обновление компонента.

Сохраним исправленный код, дождемся окончания его трансляции и пометим второе дело как выполненное. Перечень дел тотчас обновится, и его вторая позиция будет выведена зачеркнутой. Приложение работает!

Теперь займемся удалением дел. Добавим в компонент `App` вложенную функцию `del(<идентификатор>)`, которая удалит дело с заданным *идентификатором*, и передадим эту функцию компоненту `TodoList` через проп `del`.

3. Добавим в компонент `App` необходимый код:

```
export default function App() {
    . . .
    const del = key => {
        const newTodos = todos.filter(current => current.key !== key);
        setTodos(newTodos);
    };

    return (
        . . .
                ~~<TodoList list={todos} setDone={setDone} />~~
                <TodoList list={todos} setDone={setDone} del={del} />
        . . .
    );
}
```

В теле функции `del()` получаем копию массива дел, не содержащую дела с указанным идентификатором, использовав метод `filter()`. После чего изменяем состояние, занося в него полученную копию массива.

В компоненте перечня дел используем функцию из пропа `del` для обработки щелчка на кнопке **Удалить**.

4. Добавим в компонент `TodoList` требуемый код:

```
export default function TodoList(props) {
    return (
        . . .
                                <button
                                    className="button is-danger"
                                    title="Удалить"
                                    onClick={() => props.del(item.key)}
                                >
                                    &#9746;
                                </button>
        . . .
    );
}
```

Внимание!

Не забываем сохранять файлы после выполнения очередного упражнения.

Попробуем удалить какое-либо дело. Если мы не допустили ошибок в коде, у нас получится.

Настала пора заняться функциональностью создания новых дел.

1.7. Работа с веб-формами

Для добавления нового дела создадим компонент `TodoAdd`. Он будет выводиться под компонентом `TodoList`, получит от компонента-родителя `App` проп `add` с функцией, добавляющей новое дело в массив, и вызовет эту функцию, когда посетитель нажмет кнопку создания дела.

Сначала объявим в компоненте `App` вложенную функцию `add(<дело>)`, которая добавит в массив новое *дело*, представленное простым объектом.

1. Добавим в компонент `App` объявление вложенной функции `add()`:

```
export default function App() {
    . . .
    const add = deed => {
        setTodos([...todos, deed]);
    };
    . . .
}
```

Мы создаем копию массива дел, добавляем в нее новое дело, полученное с параметром, и заносим эту копию в состояние — и все в одной строке кода!

Позже мы передадим эту функцию компоненту `TodoAdd` через проп `add`.

Компонент `TodoAdd` выведет на страницу веб-форму с тремя элементами управления: полем ввода — для заголовка создаваемого дела, областью редактирова-

ния — для необязательного текстового описания и поля для выбора графического файла с иллюстрацией, также необязательной.

2. Создадим модуль src\TodoAdd.js и сохраним в нем код компонента `TodoAdd` из листинга 1.4.

Листинг 1.4. Компонент добавления нового дела `TodoAdd` (src\TodoAdd.js)

```
import { useState } from 'react';

export default function TodoAdd(props) {
    const [title, setTitle] = useState(''); // ..........................  1
    const [desc, setDesc] = useState('');
    const [image, setImage] = useState('');

    const handleImageChange = evt => { // ..............................  6
        const cFiles = evt.target.files;
        if (cFiles.length > 0) {
            const fileReader = new FileReader();
            fileReader.onload = () => {
                setImage(fileReader.result);
            }
            fileReader.readAsDataURL(cFiles[0]);
        } else
            setImage('');
    };

    const handleFormSubmit = evt => { // ................................  7
        evt.preventDefault();
        const newDeed = {title, desc, image, done: false};
        const date = new Date();
        newDeed.createdAt = date.toLocaleString();
        newDeed.key = date.getTime();
        props.add(newDeed);
        evt.target.reset();
    };

    const handleFormReset = () => { // ..................................  8
        setTitle('');
        setDesc('');
        setImage('');
    };

    return (
        <section>
            <h1>Создание нового дела</h1>
            <form onSubmit={handleFormSubmit} onReset={handleFormReset}>
```

```
                <div className="field"> // ..............................  9
                    <label className="label">Заголовок</label>
                    <div className="control">
                        <input className="input" value={title} // .......  2
                         onChange={e => setTitle(e.target.value)} /> // .  3
                    </div>
                </div>
                <div className="field">
                    <label className="label">Примечание</label>
                    <div className="control">
                        <textarea className="textarea" value={desc} // ..  4
                         onChange={e => setDesc(e.target.value)} /> // ..  5
                    </div>
                </div>
                <div className="field">
                    <div className="file"> // ........................... 10
                        <label className="file-label">
                            <input className="file-input"
                                   type="file" accept="image/*"
                                   onChange={handleImageChange} />
                            <span className="file-cta">
                                <span className="file-label">
                                    Графическая иллюстрация...
                                </span>
                            </span>
                        </label>
                    </div>
                </div>
                <div className="field is-grouped is-grouped-right"> // .. 11
                    <div className="control">
                        <input type="reset"
                               className="button is-warning is-light"
                               value="Сброс" />
                    </div>
                    <div className="control">
                        <input type="submit" className="button is-primary"
                               value="Создать дело" />
                    </div>
                </div>
            </form>
        </section>
    );
}
```

Обработка элементов управления в React выполняется следующим образом. На каждый элемент управления создается отдельное состояние, хранящее актуальное значение, занесенное в этот элемент управления. В самом элементе выво-

дится текущее значение состояния. Занесение нового значения в состояние производится в обработчике события `change` элемента. Такой подход упрощает получение значений, введенных в элементы управления (их можно извлечь из состояний), и позволяет обрабатывать их в процессе ввода (например, приводить буквы введенной строки к нижнему регистру).

Итак, для хранения значений из трех элементов управления, присутствующих в нашей веб-форме, мы создали три состояния: `title`, `desc` и `image` (поз. 1 в листинге 1.4).

В поле ввода, предназначенном для набора заголовка дела, мы выведем текущее значение соответствующего состояния (поз. 2). В обработчике события `change`, возникающего после изменения содержимого поля и потери им фокуса ввода, занесем содержимое поля в это состояние, тем самым обновив его (поз. 3).

Аналогично обрабатывается область редактирования, в которую заносится описание дела (поз. 4 и 5).

У поля выбора файла обработчик события `change` довольно сложен, и его лучше оформить в виде отдельной функции `handleImageChange()` (поз. 6). В этой функции мы загружаем файл, выбранный в поле, преобразуем его содержимое в data URL и заносим в соответствующее состояние.

В функции `handleFormSubmit()` (поз. 7), обработчике события `submit` веб-формы, будет выполняться создание нового дела. Сначала отключаем обработку события по умолчанию (в противном случае веб-обозреватель отправит данные из веб-формы той же странице, что вызовет ее перезагрузку с потерей всех данных, включая созданные посетителем дела). Далее создаем простой объект, содержащий заголовок, описание дела, иллюстрацию к нему и пометку о том, что дело не выполнено. Добавляем в созданный ранее простой объект строку с текущими датой и временем и идентификатор дела. Наконец, вызываем полученную с пропом `add` функцию, создающую новое дело, передаем этой функции созданный ранее простой объект и сбрасываем веб-форму, вызвав у нее метод `reset()`.

Функция `handleFormReset()` (поз. 8), обработчик события `reset` веб-формы, очистит все три состояния, занеся в них пустые строки. Если этого не сделать, элементы управления даже после сброса веб-формы будут содержать ранее введенные в них значения.

CSS-фреймворк Bulma позволяет создавать хорошо выглядящие веб-формы, не прибегая к сложной HTML-разметке. Чтобы создать элемент управления (поле ввода, область редактирования, список) с надписью, достаточно добавить в HTML-код блок (тег `<div>`) со стилевым классом `field` (поз. 9) и поместить в него:

- надпись (тег `<label>`) — со стилевым классом `label`;
- блок — со стилевым классом `control`, — в который помещается сам элемент управления.

У полей ввода указывается стилевой класс `input` (поз. 2), а у областей редактирования — стилевой класс `textarea` (поз. 4).

Bulma позволяет создать поле для выбора файла, имеющее вид кнопки. Для этого применяется специфическая HTML-разметка, которую можно увидеть в поз. 10.

Чтобы вывести кнопки добавления дела и сброса по горизонтали, выровняв их по правому краю веб-формы, мы использовали блок (поз. 11) со стилевыми классами `field`, `is-grouped` (размещение кнопок по горизонтали) и `is-grouped-right` (выравнивание по правому краю). Внутри поместили блоки со стилевым классом `control` с кнопками, у которых указали стилевые классы `button`. У кнопки отправки данных дополнительно указали стилевой класс `is-primary` (зеленоголубой цвет — чтобы выделить кнопку визуально), у кнопки сброса — стилевые классы `is-warning` и `is-light` (желтый цвет и отсутствие фона — чтобы кнопка меньше бросалась в глаза).

3. Вставим в компонент `App` выражение, выводящее только что написанный компонент `TodoAdd` под перечнем дел:

```
. . .
import TodoAdd from './TodoAdd.js';
. . .
export default function App() {
    . . .
    return (
        . . .
            <main className="content px-6 py-6">
                <TodoList . . . />
                <TodoAdd add={this.add} />
            </main>
        . . .
    );
}
```

Если мы не допустили ошибок, под перечнем дел появится веб-форма для создания нового дела (рис. 1.4). Проверим, работает ли она, добавив пару произвольных дел.

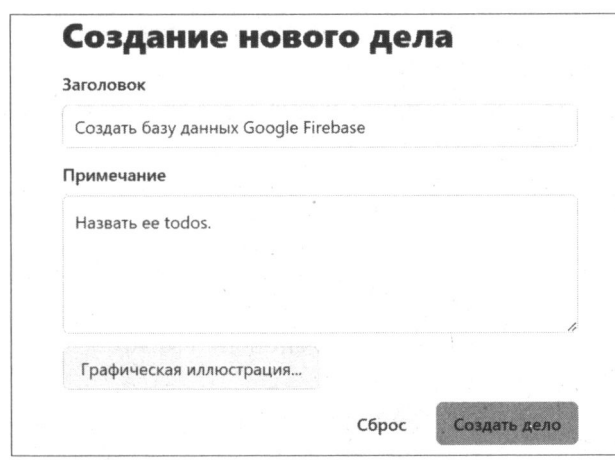


Рис. 1.4. Веб-форма для создания нового дела

К настоящему моменту наше приложение имеет всего один экран, на котором отображается перечень дел и веб-форма для ввода нового дела. В результате интерфейс приложения занимает слишком много места на экране, что не понравится тем, кто будет работать с ним на мобильных устройствах.

Поэтому мы разобьем интерфейс приложения на несколько более компактных экранов и реализуем навигацию между ними.

Остановим работу проекта, поскольку предстоят значительные переделки.

1.8. Навигация. Библиотека React Router. Маршрутизатор, маршрут и выпуск

Теория

Навигация — это перемещение с одной страницы сайта на другую в результате, например, щелчков на гиперссылках. Применительно к одностраничным веб-приложениям это перемещение с одного экрана на другой.

Поскольку React-приложение состоит из одной-единственной страницы, адрес хоста, входящий в состав интернет-адреса, в процессе навигации не меняется. Меняется только *путь* — часть интернет-адреса, располагающаяся после адреса хоста (в традиционных многостраничных сайтах путь указывает на загружаемый файл). Например, в интернет-адресе **http://someapp.ru/todos/1627298956192** путь — **/todos/1627298956192**.

Чтобы реализовать навигацию в приложении, прежде всего следует записать список маршрутов. *Маршрут* — это особый компонент, который связывает путь, совпадающий с заданным шаблоном (*шаблонным путем*), с определенным компонентом (называемым *целевым*, практически всегда это компонент-экран).

На основе списка маршрутов создается *маршрутизатор* — подсистема, которая отслеживает изменение интернет-адреса при переходе по какой-либо гиперссылке, выбирает маршрут, чей шаблонный путь совпадает с извлеченным из текущего интернет-адреса (*текущим путем*), иначе говоря, *совпавший маршрут*, и выводит компонент, записанный в совпавшем пути.

Реализовать навигацию в React-приложении удобно с помощью дополнительной библиотеки React Router.

Практика

Реализуем в нашем приложении навигацию, превратив компоненты `TodoList` и `TodoAdd` в отдельные экраны. Пусть компонент `TodoList` выводится при переходе по пути / (слеш, «корень» приложения), а компонент `TodoAdd` — при переходе по пути **/add**.

1. Установим библиотеку React Router, набрав в консоли команду:

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

Начнем с написания списка маршрутов. При его написании учтем, что компоненты `TodoList` и `TodoAdd` должны выводиться внутри компонента `App`. Список маршрутов вынесем в отдельный модуль src\routes.js.

2. Создадим модуль src\routes.js и сохраним в нем код списка маршрутов из листинга 1.5.

Листинг 1.5. Список маршрутов (src\routes.js)

```
import { createRoutesFromElements, createBrowserRouter, Route } from
       'react-router-dom';

import App from './App.js';
import TodoList from './TodoList.js';
import TodoAdd from './TodoAdd.js';

import todos from './todos.js';

const router = createBrowserRouter( // ..................................  5
    createRoutesFromElements( // ........................................  4
        <Route path='/' element={<App />}> // ............................  1
            <Route index={true} element={<TodoList list={todos} /> // ...  3
            <Route path='add' element={<TodoAdd />} /> // ................  2
        </Route>
    )
);
export default router; // ...............................................  6
```

Маршрут создается компонентом `Route`, объявленным в модуле `react-router-dom` (поз. 1 в листинге 1.5). Шаблонный путь записывается в пропе `path` этого компонента, а JSX-код, выводящий целевой компонент, — в пропе `element`.

Чтобы указать маршрутизатору React Router, что все компоненты-экраны нашего приложения должны выводиться в компоненте `App`, мы создали своего рода иерархию маршрутов. Рассмотрим ее.

Сначала мы записали *родительский* маршрут, который содержит два других маршрута — *дочерних* (поз. 1). В пропе `path` родительского маршрута указали префикс, с которого начинаются шаблонные пути дочерних маршрутов, — символ /. А в пропе `element` родительского маршрута записали код, выводящий компонент приложения `App`, в котором должны выводиться компоненты-экраны `TodoList` и `TodoAdd`, указанные в дочерних маршрутах (поз. 2 и 3).

В дочернем маршруте, выводящем компонент-экран добавления дела `TodoAdd`, мы указали шаблонный путь **add** — уже без префикса / (поскольку префикс был задан в родительском маршруте) — см. поз. 2. А в дочернем маршруте, выводящем компонент-экран перечня дел `TodoList`, мы указали, что его шаблонный путь совпадает с таковым у родительского маршрута, задав проп `index` со значением `true` (поз. 3).

В результате при переходе по пути / будет выведен компонент App, содержащий компонент TodoList, а при переходе по пути **/add** — компонент App с компонентом TodoAdd.

У компонента TodoList мы указали проп list, присвоив ему массив дел (поз. 3). Это сделано на время, только чтобы проверить компонент в работе (позже мы реализуем передачу массива дел целевому компоненту более удобным способом — с помощью загрузчика).

Готовый список маршрутов мы «пропустили» через функцию createRoutesFromElements() из модуля react-router-dom, которая преобразует набор компонентов Route в представление, пригодное к созданию на его основе маршрутизатора (поз. 4). Полученное представление мы передали функции createBrowserRouter() из того же модуля, которая создаст на его основе и вернет в качестве результата готовый маршрутизатор (поз. 5).

Константу с маршрутизатором мы экспортировали — она нам скоро понадобится (поз. 6).

Теперь следует указать React, чтобы он выводил в блоке с якорем root, присутствующим на странице public\index.html, не компонент приложения App, а особый компонент, называемый *провайдером маршрутизации*, задав в нем маршрутизатор, созданный только что. Для этого придется исправить стартовый модуль src\index.js, который инициализирует веб-приложение.

3. Исправим код стартового модуля src\index.js следующим образом:

```
. . .
import { RouterProvider } from 'react-router-dom';
~~import App from './App';~~
import router from './routes.js';
. . .
const root = ReactDOM.createRoot(document.getElementById('root')); // ...  1
root.render( // ..................................................  2
  <React.StrictMode> // ..........................................  4
    ~~<App />~~
    <RouterProvider router={router} /> // ........................  3
  </React.StrictMode>
);
```

Присутствующий в модуле код ищет в составе страницы элемент с якорем root, преобразует его в объект, способный выводить React-компоненты (поз. 1 в представленном коде), и выводит в нем компонент приложения, заключенный в компонент StrictMode (поз. 2). Нам следует заменить компонент приложения провайдером маршрутизации.

Провайдер маршрутизации представляется компонентом RouterProvider, объявленным в модуле react-router-dom (поз. 3). В пропе router указывается используемый маршрутизатор.

Компонент StrictMode из модуля react (поз. 4) задействуется при отладке и упрощает выявление ряда ошибок. Мы познакомимся с ним в *главе 23*.

На текущий момент компонент приложения выводит оба компонента: `TodoList` и `TodoAdd`. Их следует удалить, заменив *выпуском* — пометкой, в каком месте маршрутизатору следует выводить целевые компоненты.

4. Исправим код компонента `App`, как показано далее:

```
. . .
import { Outlet } from 'react-router-dom';
~~import TodoList from './TodoList.js';~~
~~import TodoAdd from './TodoAdd.js';~~
. . .
export default function App() {
    . . .
    return (
        . . .
            <main className="content px-6 py-6">
                ~~<TodoList list={todos} setDone={setDone} del={del} />~~
                ~~<TodoAdd add={add} />~~
                <Outlet />
            </main>
        . . .
    );
}
```

Выпуск представляется компонентом `Outlet` из модуля `react-router-dom`.

Запустим проект. Веб-обозреватель сразу выведет перечень дел.

Чтобы попасть на веб-форму создания нового дела, понадобится создать гиперссылку, указывающую на компонент `TodoAdd`.

Также следует создать гиперссылку на компонент `TodoList` — на случай, если пользователь передумает создавать новое дело и решит вернуться к перечню дел. Создадим эту гиперссылку из надписи «Todos», которая выводится в обозначении бренда на полосе навигации (см. рис. 1.3).

5. Добавим в компонент `App` код, создающий гиперссылки:

```
. . .
~~import { Outlet } from 'react-router-dom';~~
import { Outlet, NavLink } from 'react-router-dom';
. . .
export default function App() {
    . . .
    return (
        <div className="container">
            <nav className="navbar is-light">
                <div className="navbar-brand">
                    ~~<span className="navbar-item is-uppercase">~~ // ..... 1
                    <NavLink // ..... 2
                        to="/" // ..... 3
```

```
                        className={({ isActive }) => // ..............  4
                            'navbar-item is-uppercase' +
                            (isActive ? ' is-active' : '')
                        }
                    >
                        Todos
                    </NavLink>
                    </span>
                </div>
                <div className="navbar-menu"> // ......................  5
                    <div className="navbar-start"> // ..................  6
                        <NavLink // ....................................  7
                            to="/add"
                            className={({ isActive }) =>
                                'navbar-item' +
                                (isActive ? ' is-active' : '')
                            }
                        >
                            Создать дело
                        </NavLink>
                    </div>
                </div>
            </nav>
            . . .
        </div>
    );
}
```

В блоке (теге `<div>`) со стилевым классом `navbar-brand`, в котором записано обозначение бренда, мы убрали не нужный больше тег `<span>` (поз. 1 в приведенном коде) и заменили его компонентом `NavLink`, объявленным в модуле `react-router-dom` и реализующим гиперссылку React Router (поз. 2). Путь, по которому будет выполнен переход при щелчке на такой гиперссылке (*целевой путь*), указывается в пропе `to` (поз. 3).

Особенность гиперссылки `NavLink` в том, что она «умеет» сверять текущий путь с записанным в ней целевым и в случае их совпадения помечать себя как *активную* (указывающую на текущий путь). У активной гиперссылки, реализуемой этим компонентом, можно указать другое оформление. Сейчас выясним, как это делается.

В пропе `className` компонента `NavLink` вместо строки со стилевыми классами можно указать функцию. В качестве параметра она принимает простой объект со свойством `isActive`, которое хранит `true`, если текущая гиперссылка является активной, и `false` — в противном случае.

Мы присвоили пропу `className` нашей гиперссылки функцию (поз. 4), которая задает у любой гиперссылки стилевые классы `navbar-item` и `is-uppercase` (тем самым создавая обычное, «неактивное» оформление), а у активной — дополни-

тельно стилевой класс `is-active` (визуально выделяющий активную гиперссылку).

Для создания собственно меню навигации следует поместить в полосу навигации блок со стилевым классом `navbar-menu` (поз. 5). Он должен находиться непосредственно после блока со стилевым классом `navbar-brand`, в котором записано обозначение бренда.

Bulma позволяет разделить *меню навигации* на две части, выводящиеся соответственно слева и справа.

Левая часть этого меню создается блоком со стилевым классом `navbar-start` (поз. 6). Туда мы поместили гиперссылку **Создать дело** (поз. 7), указывающую на экран добавления нового дела.

Здесь возникнет еще одна проблема. При ширине экрана (или окна веб-обозревателя) менее 1024 пикселов (т. е. при просмотре страницы на мобильных устройствах) Bulma, во-первых, формирует вертикальное (а не горизонтальное) меню навигации, а во-вторых, скрывает это меню. Выводить его на экран разработчику приложения придется самостоятельно. Обычно меню навигации выводится по щелчку на *кнопке-гамбургере*, несущей три горизонтальные черточки[1], которая появляется также лишь при ширине экрана менее 1024 пикселов.

Нам понадобится создать в полосе навигации кнопку-гамбургер, написать обработчик события щелчка на ней и реализовать в этом обработчике вывод и скрытие меню навигации.

6. Добавим в компонент `App` код, реализующий описанную функциональность:

```
export default function App() {
    . . .
    const [showMenu, setShowMenu] = useState(false); // .................  2
    . . .
    const handleBurgerClick = evt => { // ..............................  3
        evt.preventDefault();
        setShowMenu(!showMenu);
    };

    return (
        <div className="container">
            <nav className="navbar is-light">
                <div className="navbar-brand">
                    <NavLink . . . />
                    . . .
                    <a href="/" // .....................................  1
                        className={showMenu ? // .......................  6
                            'navbar-burger is-active' :
                            'navbar-burger'}
```

[1] Которые отдаленно похожи на гамбургер.

```
                        onClick={handleBurgerClick} // ..................  4
                    >
                        <span></span>
                        <span></span>
                        <span></span>
                        <span></span>
                    </a>
                </div>
                <div className="navbar-menu">
                <div className={showMenu ? // ..........................  7
                                'navbar-menu is-active' :
                                'navbar-menu'}
                    onClick={handleBurgerClick} // ......................  5
                >
                    . . .
                </div>
            </nav>
            . . .
        </div>
    );
  }
}
```

Кнопка-гамбургер в стиле фреймворка Bulma создается тегом `<a>` со стилевым классом `navbar-burger` и четырьмя пустыми встроенными контейнерами (тегами `<span>`) в качестве содержимого (поз. 1 в приведенном коде). Эта кнопка помещается в блок с обозначением бренда.

Чтобы реализовать вывод и скрытие меню навигации, создадим в компоненте новое состояние `showMenu` (поз. 2), которое будет хранить значение `false` (меню скрыто) или `true` (меню выведено на экран). И напишем обработчик события `click` кнопки-гамбургера, который станет изменять состояние `showMenu`, инвертируя его значение (поз. 4). В результате, если ранее меню навигации было скрыто, то оно появится на экране, и наоборот.

Этот обработчик события `click` следует привязать, помимо кнопки-гамбургера (поз. 4), также к меню навигации (поз. 5). Это необходимо для того, чтобы меню навигации скрывалось после выбора в нем какого-либо пункта.

Чтобы вывести меню навигации, достаточно указать у формирующего его блока стилевой класс `is-active`. Мы используем для этого выражение JavaScript, вставленное в JSX-код и — в зависимости от значения состояния `showMenu` — указывающее у меню разные наборы стилевых классов (поз. 6).

При выводе меню навигации также следует записать стилевой класс `is-active` в кнопке-гамбургере (поз. 7). В результате этого изображение «гамбургера» на кнопке сменится крестиком — универсальным символом закрытия.

Посмотрим, что показывает веб-обозреватель. Уменьшим ширину его окна, пока в правой части полосы навигации не появится кнопка-гамбургер (рис. 1.5). Щелк-

нем на ней и выберем в появившемся меню навигации (рис. 1.6) пункт **Создать дело**, чтобы переключиться на экран создания нового дела. Вернемся на экран перечня дел щелчком на гиперссылке «Todos».

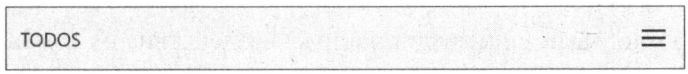


Рис. 1.5. Полоса навигации с кнопкой-гамбургером в правой части

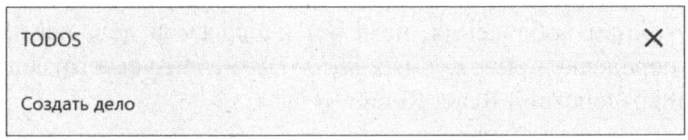


Рис. 1.6. Полоса навигации после щелчка на кнопке-гамбургере — меню навигации выведено

Попробуем увеличить ширину окна веб-обозревателя, пока меню навигации не будет полностью выведено на экран (рис. 1.7). Отметим, что кнопка-гамбургер при этом пропадет.

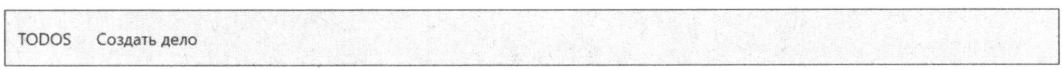


Рис. 1.7. Полоса навигации с меню навигации

Однако если мы попытаемся добавить новое дело, получим ошибку. Наделяя приложение средствами навигации, мы «сломали» механизм добавления, пометки и удаления дел.

И сейчас начнем «чинить» его.

1.9. Загрузчики и воздействия React Router. Перенаправление

Теория

Маршрутизатор, реализуемый библиотекой React Router, полностью имитирует навигацию по обычным, многостраничным сайтам и веб-приложениям. Так, при щелчке на гиперссылке он имитирует отправку и обработку полноценного клиентского запроса, выполненного HTTP-методом GET, а при щелчке на кнопке отправки данных в веб-форме — запроса методом, который указан в параметрах создаваемого запроса: POST, PUT, PATCH или DELETE.

А еще он позволяет при получении очередного клиентского запроса по заданному шаблонному пути выполнить указанный фрагмент программного кода, относящийся к одной из двух разновидностей:

- *загрузчик* — выполняется при получении клиентского запроса HTTP-методом GET. Обычно производит загрузку данных, преобразование их к виду, предназначенному для вывода, и передачу их целевому компоненту, записанному в соответствующем маршруте;

- *воздействие React Router* — выполняется при получении запроса методом POST, PUT, PATCH или DELETE. Обычно вносит заданные изменения в данные: добавляет, изменяет или удаляет отдельные сущности. Может выдавать целевому компоненту какие-либо данные для вывода на экран.

Или, совсем коротко: загрузчик читает данные, воздействие их изменяет.

Практика

«Исправим» механизм добавления, пометки и удаления дел, невзначай «сломанный» нами при переделке приложения в *разд. 1.8*. Реализуем его заново — на основе загрузчиков и воздействий React Router.

Сначала, так сказать, подготовим почву, удалив ненужный код. Из компонента `App` необходимо удалить, во-первых, объявления вложенных функций `add()`, `setDone()` и `del()`, которые в настоящее время никак не используются. Еще следует удалить выражение, которое создает состояние `todos`, хранящее массив дел и также ненужное.

1. Удалим из компонента `App` ненужный код:

```
. . .
import initialTodos from './todos.js';

export default function App() {
    const [todos, setTodos] = useState(initialTodos);
    const [showMenu, setShowMenu] = useState(false);

    const add = deed => {
        setTodos([...todos, deed]);
    };

    const setDone = key => {
        . . .
    };

    const del = key => {
        . . .
    };
    . . .
}
```

2. Удалим из модуля списка маршрутов src\routes.js фрагмент JSX-кода, передающий массив дел компоненту `TodoList` через проп `list`:

```
import todos from './todos.js';

const router = createBrowserRouter(
    createRoutesFromElements(
        <Route path='/' element={<App />}>
```

```
            <Route index={true} element={<TodoList list={todos} />} />
            <Route index={true} element={<TodoList />} />
            <Route path='add' element={<TodoAdd />} />
        </Route>
    )
);
```

После этого можно приступить к написанию кода, выполняющего работу с массивом дел.

Мы напишем загрузчик и воздействие, реализовав их в виде функций:

- загрузчик `getTodos()` — извлекающий и передающий компоненту `TodoList` массив дел. Будет выполняться при переходе на упомянутый компонент, то есть по пути /;
- воздействие `addTodo()` — добавляющее новое дело в массив. Будет выполняться при имитации клиентского запроса по пути **/add** HTTP-методом POST.

  Условимся, что новое дело будем пересылать этому воздействию в виде набора POST-параметров `title` (заголовок дела), `desc` (описание) и `image` (иллюстрация), закодированного в формате `application/x-www-form-urlencoded`.

Код загрузчика и воздействия вынесем во вновь созданный модуль **src\api.js**.

3. Создадим модуль **src\api.js** и сохраним в нем код из листинга 1.6.

Листинг 1.6. Загрузчики и воздействия (src\api.js)

```
import { redirect } from 'react-router-dom';

import todos from './todos.js';

export function getTodos() { // ..................................... 1
    return todos;
}

export async function addTodo({ request }) { // ..................... 2
    const fd = await request.formData(); // ......................... 3
    const date = new Date();
    const newTodo = { // ............................................ 4
        title: fd.get('title'),
        desc: fd.get('desc'),
        image: fd.get('image'),
        done: false,
        createdAt: date.toLocaleString(),
        key: date.getTime()
    };
    todos.push(newTodo); // ......................................... 5
    return redirect('/'); // ....................................... 6
}
```

Код загрузчика `getTodos()` очень прост — он лишь возвращает массив дел, извлеченный из константы `todos`, которая объявлена в модуле src\todos.js (поз. 1 в листинге 1.6).

Код воздействия `addTodo()` несколько сложнее. Прежде всего, нам нужно получить данные, оправленные из веб-формы ввода нового дела (находятся в компоненте `TodoAdd`). Их можно получить из объекта запроса (создается на основе стандартного класса `Request`), который хранится в свойстве `request` простого объекта, получаемого воздействием в единственном параметре. Для удобства сразу деструктурируем этот объект (поз. 2).

Имея объект запроса, извлекаем из него данные, отправленные веб-формой. Эти данные представлены объектом класса `FormData`, и для их получения достаточно вызвать асинхронный метод `formData()` (поз. 3). Создаем простой объект, представляющий добавляемое дело, и «наполняем» его значениями, взятыми из полученного объекта с данными (поз. 4). После чего просто добавляем подготовленный объект дела в массив (поз. 5).

После добавления нового дела необходимо произвести принудительный переход на экран перечня дел (выполнить *перенаправление*). Делаем это, вызвав функцию `redirect()` из модуля `react-router-dom` и передав ей строку с путем `/` (поз. 6).

4. Впишем в код списка маршрутов src\routes.js готовые загрузчик и воздействие:

```
. . .
import { getTodos, addTodo } from './api.js';

const router = createBrowserRouter(
    createRoutesFromElements(
        <Route path='/' element={<App />}>
            <Route index={true} element={<TodoList />} />
            <Route index={true} element={<TodoList />}
                   loader={getTodos} /> // ..............................  1
            <Route path='add' element={<TodoAdd />} />
            <Route path='add' element={<TodoAdd />}
                   action={addTodo} /> // ..............................  2
        </Route>
    )
);
. . .
```

Загрузчик указывается в пропе `loader` (поз. 1 в приведенном коде), а воздействие — в пропе `action` (поз. 2).

Теперь нужно предписать компоненту `TodoList` получение массива дел от загрузчика `getTodos()`. Сделать это очень просто, и мы в этом убедимся сами, для чего...

5. ...внесем в код компонента `TodoList` исправления — совсем незначительные:

```
import { useLoaderData } from 'react-router-dom';

~~export default function TodoList(props) {~~
export default function TodoList() {
    const list = useLoaderData();

    return (
        <section>
            . . .
            <table className="table is-hoverable is-fullwidth">
                <tbody>
                    ~~{props.list.map(item => (~~
                    {list.map(item => (
                        <tr key={item.key}>
                            . . .
                        </tr>
                    ))}
                </tbody>
            </table>
        </section>
    );
}
```

Получить данные от загрузчика позволяет хук `useLoaderData()` из модуля `react-router-dom`. В качестве результата он возвращает само значение, выданное загрузчиком.

Осталось последнее — указать компоненту `TodoAdd` при нажатии кнопки отправки данных в веб-форме помещать введенные в нее данные в клиентский запрос и отправлять его по пути **/add** HTTP-методом POST. Сделать это также несложно.

6. Переделаем код компонента `TodoAdd` (**src\TodoAdd.js**) следующим образом:

```
. . .
import { useSubmit } from 'react-router-dom';

~~export default function TodoAdd(props) {~~
export default function TodoAdd() {
    . . .
    const submit = useSubmit(); // ..................................  1
    . . .
    const handleFormSubmit = evt => {
        evt.preventDefault();
        ~~// . . . Остальной код . . .~~
        submit({title, desc, image}, // ..............................  2
               {action: '/add', method: 'post'});
    };
    . . .
}
```

Вызвав хук `useSubmit()` из модуля `react-router-dom`, мы получим *отправитель клиентских запросов* — функцию, которая создаст клиентский запрос согласно заданным параметрам и отправляет его маршрутизатору (поз. 1 в приведенном коде).

Полученный отправитель запросов вызываем в обработчике события `submit` веб-формы (поз. 2). Первым параметром передаем простой объект, содержащий введенные в веб-форму данные и подлежащий пересылке воздействию `addTodo()`. Вторым параметром задаем простой объект с параметрами создаваемого клиентского запроса: путь **/add** и HTTP-метод POST.

Добавим в перечень три дела, в одном из них из необязательных сведений указав лишь описание, в другом — только иллюстрацию, в третьем — и описание, и иллюстрацию.

Осталось сделать еще один экран — для просмотра полных сведений о выбранном деле, включая описание и иллюстрацию. И заставить работать пометку и удаление дел.

1.10. Параметризованные маршруты

Теория

Очень часто целевому компоненту в процессе навигации приходится передавать какие-либо данные. Например, компоненту, выводящему сведения о деле, следует передать идентификатор выводимого дела — иначе он просто не «узнает», какое дело нужно вывести.

Передать данные целевому компоненту можно непосредственно в составе пути, указывающем на него. Например, чтобы компонент сведений о выбранном деле вывел дело с идентификатором, например 1627298956192, можно поместить в ссылающуюся на него гиперссылку путь такого вида: **/todos/1627298956192**. Компонент извлечет из этого пути записанный после фрагмента **/todos/** идентификатор и успешно выведет соответствующее дело.

Данные, передаваемые в составе пути (*URL-параметры*), извлекаются и отправляются целевому компоненту маршрутизатором. Чтобы маршрутизатор «знал», какая часть пути отведена под передачу URL-параметров, в соответствующий шаблонный путь необходимо вставить обозначения этих URL-параметров. Каждый URL-параметр должен иметь уникальное имя, по которому к нему можно обратиться для извлечения переданного с ним значения.

Маршрут, в шаблонном пути которого присутствуют URL-параметры, носит название *параметризованного*.

Практика

Реализуем вывод сведений о деле, выбранном посетителем. Для этого напишем компонент `TodoDetail`. *Идентификатор* выводимого дела будет передаваться этому

компоненту в составе шаблонного пути формата */<идентификатор>* в URL-параметре, который мы вставим в шаблонный путь, — дадим ему имя `key`.

После чего заставим работать функциональность пометки и удаления дел.

Объявим в модуле загрузок и воздействий загрузчик `getTodo()`, который найдет в массиве и вернет дело с указанным идентификатором.

1. Добавим в модуль загрузок и воздействий src\api.js необходимый код:

```
export function getTodo({ params }) { // ..............................  1
    const key = +params.key; // ....................................  2
    const todo = todos.find(current => current.key === key); // .........  3
    return todo; // ................................................  4
}
```

В простом объекте, передаваемом в качестве параметра загрузчику, также присутствует свойство `params`. Оно хранит простой объект, содержащий все полученные в составе пути URL-параметры. Имена свойств этого объекта совпадают с именами URL-параметров, а из значения представляют собой значения этих URL-параметров. Так, что для получения значения URL-параметра `key` достаточно обратиться к одноименному свойству этого объекта (поз. 1 в приведенном коде).

Значения всех URL-параметров представляются в виде строк. Поэтому перед поиском в массиве соответствующего дела следует преобразовать полученный идентификатор в число. Проще всего это сделать посредством оператора «унарный плюс» (поз. 2).

Далее мы ищем в массиве дело с полученным идентификатором, воспользовавшись методом `find()` массива (поз. 3), и возвращаем найденное дело (поз. 4).

Теперь можно написать сам компонент `TodoDetail`.

2. Создадим модуль src\TodoDetail.js и запишем в него код компонента `TodoDetail` из листинга 1.7.

Листинг 1.7. Компонент сведений о выбранном деле `TodoDetail` (src\TodoDetail.js)

```
import { useLoaderData } from 'react-router-dom';

export default function TodoDetail() {
    const todo = useLoaderData(); // ..................................  1

    return (
        <section>
            {todo.done &&
                <p className="has-text-success"> // ....................  2
                    Выполнено
                </p>
            }
```

```
            <h1>{todo.title}</h1>
            <p>{todo.createdAt}</p>
            {todo.desc && <p>{todo.desc}</p>}
            {todo.image && <p><img src={todo.image}
                                   alt="Иллюстрация" /></p>}
        </section>
    );
}
```

Для получения дела, возвращенного ранее написанным загрузчиком, снова применяем хук `useLoaderData()` из модуля `react-router-dom` (поз. 1 в листинге 1.7).

Если дело помечено как выполненное, компонент выведет абзац с текстом **Выполнено** (поз. 2). Стилевой класс `has-text-success`, указанный у этого абзаца, задаст тексту зелено-голубой цвет.

Далее компонент выведет заголовок, дату и время создания дела, описание и иллюстрацию, если таковые присутствуют в составе дела.

3. Добавим в модуль списка маршрутов src\routes.js маршрут, указывающий на компонент `TodoDetail`:

```
. . .
import TodoDetail from './TodoDetail.js';

~~import { getTodos, addTodo } from './api.js';~~
import { getTodos, addTodo, getTodo } from './api.js';

const router = createBrowserRouter(
    createRoutesFromElements(
        <Route path='/' element={<App />}>
            . . .
            <Route path=':key' element={<TodoDetail />}
                   loader={getTodo} />
        </Route>
    )
);
. . .
```

Обозначения URL-параметров в шаблонном пути записываются в формате `:<имя URL-параметра>`.

Осталось превратить заголовки дел, выводящиеся в компоненте `TodoList`, в гиперссылки на компонент `TodoDetail`, чтобы посетитель смог просмотреть полные сведения о выбранном деле.

4. Добавим в компонент `TodoList` код, создающий эти гиперссылки:

```
~~import { useLoaderData } from 'react-router-dom';~~
import { useLoaderData, Link } from 'react-router-dom';
```

```
export default function TodoList() {
    . . .
    return (
        . . .
                        <tr key={item.key}>
                            <td>
                                <Link to={`/${item.key}`}>
                                    {item.done && <del>{item.title}</del>}
                                    {!item.done && item.title}
                                </Link>
                            </td>
                            . . .
                        </tr>
        . . .
    );
}
```

Компонент `Link` из модуля `react-router-dom` представляет собой упрощенную разновидность описанного в *разд. 1.8* компонента `NavLink`. Гиперссылка `Link` не «умеет» проверять, является ли она активной, зато отнимает меньше системных ресурсов.

Проверим приложение, просмотрев сведения о нескольких имеющихся делах (рис. 1.8).

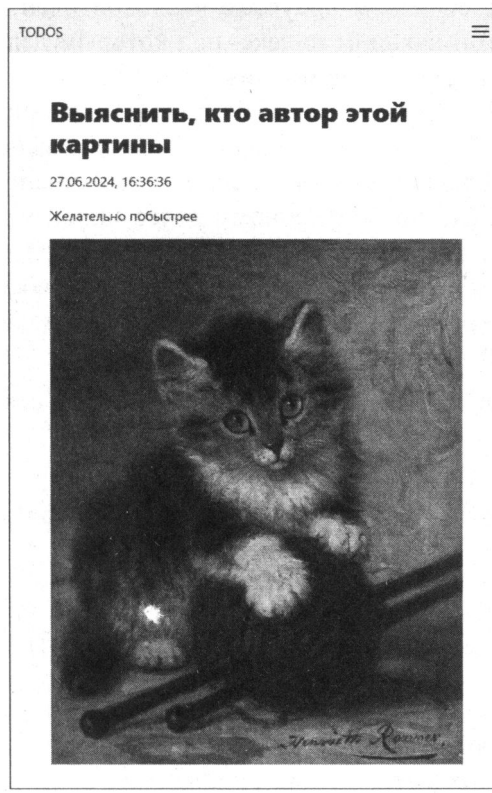


Рис. 1.8. Сведения о выбранном деле

Займемся пометкой и удалением дел. Напишем воздействие `actTodo()`, которое при получении клиентского запроса, отправленного HTTP-методом:

- PATCH — пометит дело с идентификатором, переданным в составе пути, как выполненное;
- DELETE — удалит дело с полученным идентификатором.

Идентификатор обрабатываемого дела будет передаваться в составе пути формата */<идентификатор>*. Это позволит поместить воздействие `actTodo()` в тот же маршрут, который указывает на компонент `TodoDetail`.

5. Добавим в модуль загрузок и воздействий src\api.js код, объявляющий это воздействие:

```
export function actTodo({ params, request }) {
    const key = +params.key; // ..................................... 1
    const todo = todos.findIndex(current => current.key === key); // .... 2
    if (request.method === 'PATCH') // ............................... 3
        todos[todo].done = true; // ..................................... 4
    else
        todos.splice(todo, 1); // ....................................... 5
    return redirect('/'); // ........................................... 6
}
```

Здесь мы описанным ранее способом получаем числовой идентификатор дела (поз. 1 в приведенном коде) и находим индекс, под которым дело с этим идентификатором присутствует в массиве, применив метод `findIndex()` класса `Array` (поз. 2). Обозначение HTTP-метода, которым был выполнен запрос, можно получить из свойства `method` объекта запроса. Если запрос был выполнен методом PATCH (то есть посетитель решил пометить дело как выполненное, см. поз. 3), мы извлекаем из массива дело по полученному ранее индексу и присваиваем свойству `done` этого дела значение `true` (поз. 4). В противном случае (если посетитель намеревается удалить дело) удаляем дело из массива, воспользовавшись методом `splice()` класса `Array` (поз. 5). В любом случае производим перенаправление на компонент перечня дел (поз. 6).

6. Впишем в код списка маршрутов src\routes.js написанное воздействие:

```
. . .
~~import { getTodos, addTodo, getTodo } from './api.js';~~
import { getTodos, addTodo, getTodo, actTodo } from './api.js';

const router = createBrowserRouter(
    createRoutesFromElements(
        <Route path='/' element={<App />}>
            . . .
            <Route path=':key' element={<TodoDetail />}
                ~~loader={getTodo} />~~
                loader={getTodo} action={actTodo} />
```

```
            </Route>
        )
    );
    . . .
```

И напишем в компоненте перечня дел обработчики событий щелчков на кнопках **Выполнено** и **Удалить**, которые будут вызывать это воздействие.

7. Внесем в код компонента `TodoList` необходимые исправления:

```
~~import { useLoaderData, Link } from 'react-router-dom';~~
import { useLoaderData, Link, useSubmit } from 'react-router-dom';

export default function TodoList() {
    . . .
    const submit = useSubmit(); // ..................................  1

    const handleDoneClick = key => {
        submit(null, {action: `/${key}`, method: 'patch'}); // .........  2
    };

    const handleDeleteClick = key => {
        submit(null, {action: `/${key}`, method: 'delete'}); // .........  3
    };

    return (
        . . .
                                <button
                                    className="button is-success"
                                    title="Выполнено"
                                    disabled={item.done}
                                  ~~onClick={() => props.setDone(item.key)}~~
                                onClick={() => {handleDoneClick(item.key)}}
                                >
                                    &#9745;
                                </button>
        . . .
                                <button
                                    className="button is-danger"
                                    title="Удалить"
                                    ~~onClick={() => props.del(item.key)}~~
                                 onClick={() => {handleDelClick(item.key)}}
                                >
                                    &#9746;
                                </button>
        . . .
    );
}
```

Мы получаем отправитель клиентских запросов, применив знакомый хук `useSubmit()` (поз. 1 в приведенном примере). После чего вызываем его в обработчиках событий `click` обеих кнопок (поз. 2 и 3).

Поскольку теперь нет нужды отправлять воздействию какие бы то ни было данные, в обоих случаях мы передаем отправителю запросов в первом параметре значение `null`. И не забываем в объекте с параметрами создаваемого запроса указать соответствующий шаблонный путь и HTTP-метод: в первом случае — PATCH (поз. 2), во втором — DELETE (поз. 3).

Попробуем пометить одно из дел как выполненное и удалить другое дело. Поскольку мы не допустили ошибок (так ведь?), у нас всё получится.

1.11. Вывод сообщения об ошибке 404

Наше приложение сейчас никак не обрабатывает пресловутую *ошибку 404* — переход по несуществующему пути. При попытке вывести несуществующее дело оно отобразит сообщение об ошибке, рассчитанное на программистов и ничего не говорящее обычному посетителю. Аналогичное сообщение появится при попытке перейти по любому неподдерживаемому пути — например, **/addtodo** или **/delete**. С этим нужно что-то делать — и делать прямо сейчас!

Начнем с написания компонента, который будет выводиться при попытке перейти по неподдерживаемому пути и показывать сообщение об ошибке 404. Назовем этот компонент `Error404`.

1. Создадим модуль **src\Error404.js** и запишем в него код компонента из листинга 1.8.

Листинг 1.8. Компонент сообщения об ошибке 404 `Error404` (src\Error404.js)

```
import { Link } from 'react-router-dom';

export default function Error404() {
    return (
        <section>
            <h1>Дело не существует</h1>
            <p>
                Вернитесь на <Link to="/">перечень дел</Link> и
                выберите какое-либо другое.
            </p>
        </section>
    );
}
```

Код компонента очень прост и комментариев не требует.

Запишем этот компонент в маршрут, указывающий на компонент сведений о деле.

2. Исправим код списка маршрутов src\routes.js следующим образом:

```
. . .
import Error404 from './Error404.js';
. . .
const router = createBrowserRouter(
    createRoutesFromElements(
        <Route path='/' element={<App />}>
            . . .
            <Route path=':key' element={<TodoDetail />}
                  loader={getTodo} action={actTodo} />
                  loader={getTodo} action={actTodo}
                  errorElement={<Error404 />} />
        </Route>
    )
);
. . .
```

JSX-код, выводящий компонент, который сигнализирует о возникновении любой ошибки в коде целевого компонента, загрузчика или воздействия, содержащегося в маршруте, записывается в пропе `errorElement` этого маршрута.

Осталось инициировать возникновение ошибки в коде загрузчика `getTodo()` в том случае, если дело с полученным идентификатором не существует. Для этого достаточно выполнить проверку, было ли найдено дело, и, если оно не было найдено, возбудить исключение.

3. Исправим код модуля загрузчиков и воздействий src\api.js следующим образом:

```
export function getTodo({ params }) {
    const key = +params.key;
    const todo = todos.find(current => current.key === key);
    if (!todo)
        throw new Error();
    return todo;
}
```

Откроем сведения о каком-либо деле, внесем какое-нибудь исправление в идентификатор дела, присутствующий в составе интернет-адреса, который выводится в панели веб-обозревателя, и выполним переход по исправленному интернет-адресу. Поскольку дела с таким идентификатором, скорее всего, нет, мы увидим сообщение об ошибке 404 (рис. 1.9).

В этой главе мы разработали простое React-приложение списка планируемых дел. К настоящему моменту оно хранит список дел в оперативной памяти, что крайне ненадежно, — список будет утерян после закрытия веб-обозревателя или даже при перезагрузке страницы. Однако в следующей главе мы это исправим.

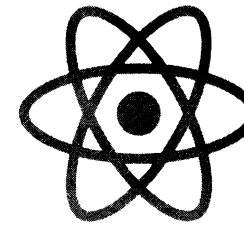
Готовое веб-приложение...

...хранится в папке todos-1-0 сопровождающего книгу электронного архива (см. *приложение 3*).



Рис. 1.9. Сообщение об ошибке 404

ГЛАВА 2



Список планируемых дел 2.0

https://t.me/it_boooks/2

React-приложение, написанное в предыдущей главе, имеет ключевой недостаток — список планируемых дел хранится в оперативной памяти и теряется при закрытии веб-обозревателя или даже перезагрузке страницы. Поэтому, чтобы приложение стало пригодным к практическому использованию, для этого списка следует выбрать более надежное хранилище.

Переделаем приложение так, чтобы оно хранило свои данные в базе данных на платформе Google Firebase. Помимо этого, сделаем приложение многопользовательским, реализовав в нем хранение отдельных списков дел для каждого из зарегистрированных посетителей.

2.1. Создание базы данных Google Firebase и подготовка приложения к работе с ней

Google Firebase — это популярная интернет-служба, предназначенная для разработчиков приложений. В ее состав входят: документная СУБД[1], облачное хранилище файлов, веб-хостинг для статических сайтов и др. Фактически Firebase позволяет разрабатывать *бессерверные* клиентские веб-приложения, не требующие написания отдельного бэкенда.

Firebase предоставляет, помимо ряда платных тарифных планов (различающихся ценой и объемом предоставляемых услуг), еще и бесплатный. Для начинающих программистов и разработчиков некоммерческих приложений это наилучший вариант.

Создадим в службе Firebase базу данных `todos`, в которой будут храниться списки планируемых дел, и подсистему разграничения доступа с регистрацией посетителей по адресу электронной почты и паролю. Также установим в проекте `todos` кли-

[1] Вообще-то, там имеются две СУБД: Realtime Database и Cloud Firestore. Мы будем использовать Realtime Database, так как работать с ней проще, а ее возможностей для нашего простого приложения достаточно.

ентскую библиотеку для взаимодействия с Firebase и подготовим приложение для работы с этой службой.

1. Перейдем по интернет-адресу **https://firebase.google.com/**, чтобы попасть на «домашний» сайт Google Firebase.
2. Щелкнем на гиперссылке **Go to console**, расположенной в правом верхнем углу открывшейся веб-страницы, чтобы попасть на страницу консоли — центра управления всеми инструментами, предоставляемыми Firebase.
3. Если не сделали этого ранее — выполним вход в интернет-службы Google.

   Сначала следует создать новый *серверный проект* Firebase (не путать с проектом React-приложения), который станет «вместилищем» для наших базы данных и подсистемы разграничения доступа.
4. Щелкнем на белой кнопке **Create a project**, расположенной на странице консоли, чтобы создать новый серверный проект.
5. Занесем в веб-форму, расположенную на открывшейся странице (рис. 2.1), следующие данные:

   - **Project name** — имя создаваемого проекта: `todos`.

     Если введенное имя совпадает с именем какого-либо из уже существующих проектов, созданных другими клиентами Firebase, для достижения уникальности заданного имени служба добавит к нему набор произвольных симво-

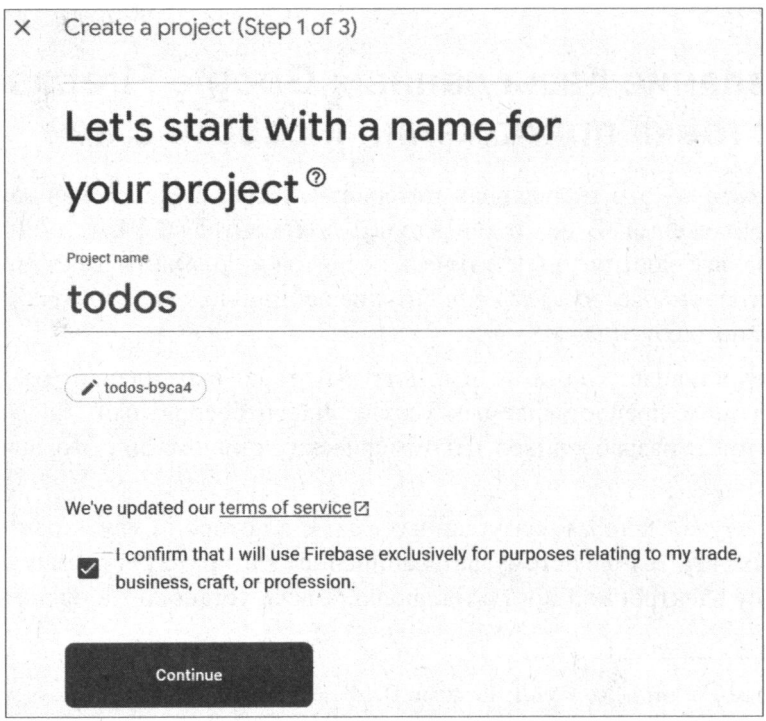


Рис. 2.1. Создание нового проекта Google Firebase, этап 1

лов. Сгенерированное таким образом заведомо уникальное имя будет выведено под полем ввода **Project name**. Так, после ввода автором имени `todos`, явно уже «занятого», служба превратила его в `todos-b9ca4`;

- **I confirm that I will use Firebase exclusively for purposes relating to my trade, business, craft, or profession** (может отсутствовать) — установим этот флажок, соглашаясь с правилами пользования Firebase (прочитать которые можно, перейдя по гиперссылке **terms of service**, расположенной в надписи к флажку).

6. Нажмем на кнопку **Continue** для перехода ко второму этапу создания проекта.

   Очередная страница (рис. 2.2) порекомендует задействовать в создаваемом проекте услугу Google Analytics, которая предлагает сбор статистики работы базы данных, анализ действий посетителей и др. На взгляд автора, при использовании лишь базы данных и разграничения доступа эта услуга не нужна.

7. Сбросим выключатель **Enable Google Analytics for this project**, чтобы отказаться от услуги Google Analytics.

8. Нажмем на кнопку **Create project** — для запуска процесса создания проекта.

   Придется подождать около минуты, пока Google создает проект.

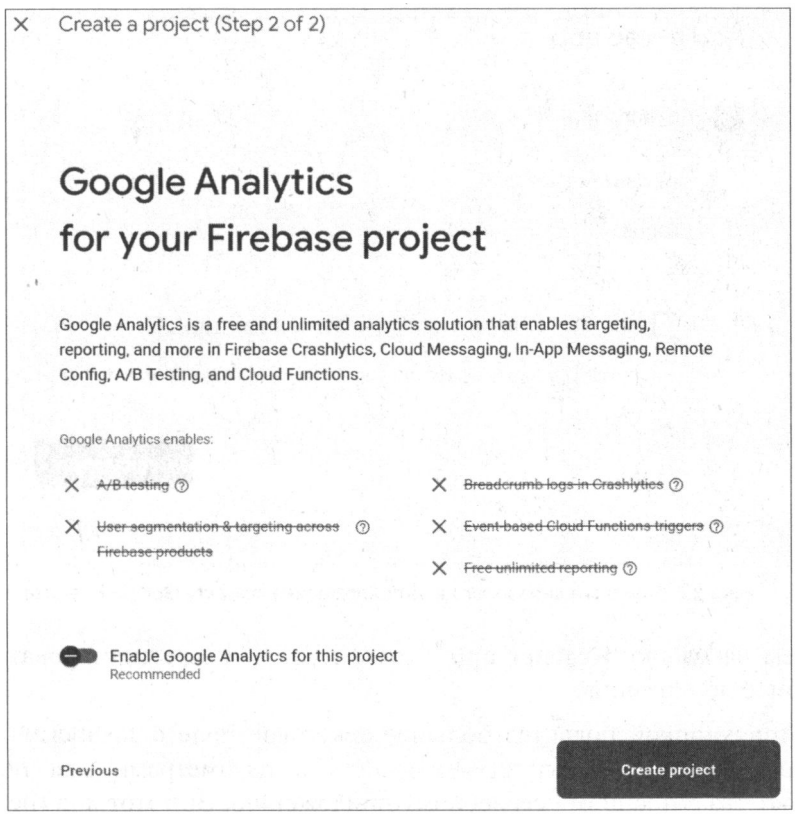


Рис. 2.2. Создание нового проекта Google Firebase, этап 2

Внимание!

Любой вновь созданный серверный проект Google Firebase по умолчанию связывается с бесплатным тарифным планом Spark.

9. Нажмем на кнопку **Continue** на следующей странице, чтобы вернуться в консоль.

   Будет открыта страница только что созданного проекта.

   Теперь необходимо создать и зарегистрировать в только что созданном проекте *серверное веб-приложение*, которое станет своего рода представлением нашего клиентского React-приложения в Firebase.

10. Нажмем на круглую кнопку **</>** в верхнем левом углу страницы, чтобы открыть страницу создания серверного приложения.

11. Занесем в веб-форму сведений о создаваемом веб-приложении (рис. 2.3) следующие данные:

    - **App nickname** — имя приложения: `Todos`;
    - **Also set up Firebase Hosting for this app** — не станем устанавливать этот флажок, т. к. не собираемся публиковать свое приложение на хостинге Firebase.

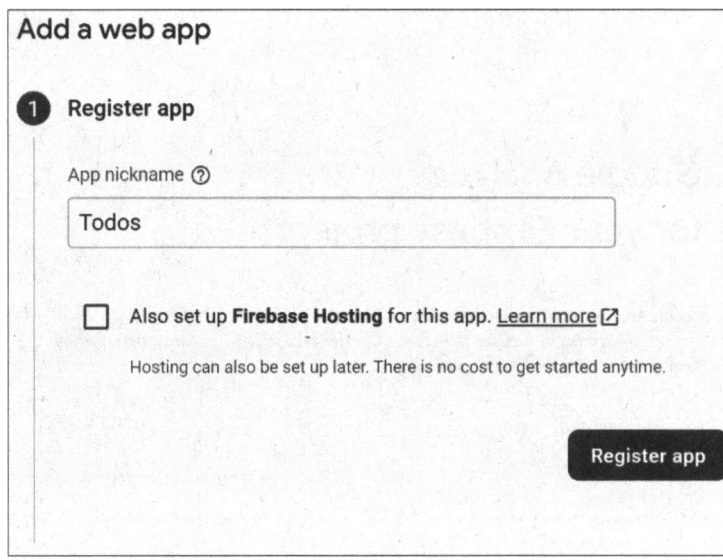


Рис. 2.3. Привязка серверного веб-приложения к проекту Google Firebase

12. Нажмем на кнопку **Register app**, чтобы создать и зарегистрировать в проекте серверное приложение.

    Под этой кнопкой появится большое текстовое поле с JavaScript-кодом, создающим простой объект `firebaseConfig` с параметрами для подключения к только что созданному серверному приложению. Вот этот код (реальные значения параметров заменены многоточиями из соображений безопасности):

```
const firebaseConfig = {
    apiKey: . . .,
    authDomain: . . .,
    projectId: . . .,
    storageBucket: . . .,
    messagingSenderId: . . .,
    appId: . . .
};
```

13. Скопируем этот код в буфер обмена.
14. Создадим в проекте нашего приложения модуль src\firebase.js и сохраним в нем скопированный ранее код.
15. Нажмем на кнопку **Continue to console**, чтобы вернуться в консоль.

    Теперь можно создать базу данных Realtime Database.

> ***Внимание!***
> Бесплатный тарифный план Spark позволяет создать в серверном проекте лишь одну базу данных.

16. Развернем в расположенной слева панели навигации группу пунктов **Build** и щелкнем на находящемся в ней пункте **Realtime Database**, чтобы переключиться в раздел, предназначенный для управления базой данных.
17. Нажмем на белую кнопку **Create Database**, чтобы открыть диалоговое окно создания базы данных.

    На первой странице этого диалогового окна (рис. 2.4) задается место хранения базы данных. Автор рекомендует выбрать место поближе.
18. Выберем в раскрывающемся списке **Realtime Database location** пункт **Belgium** (Бельгия).

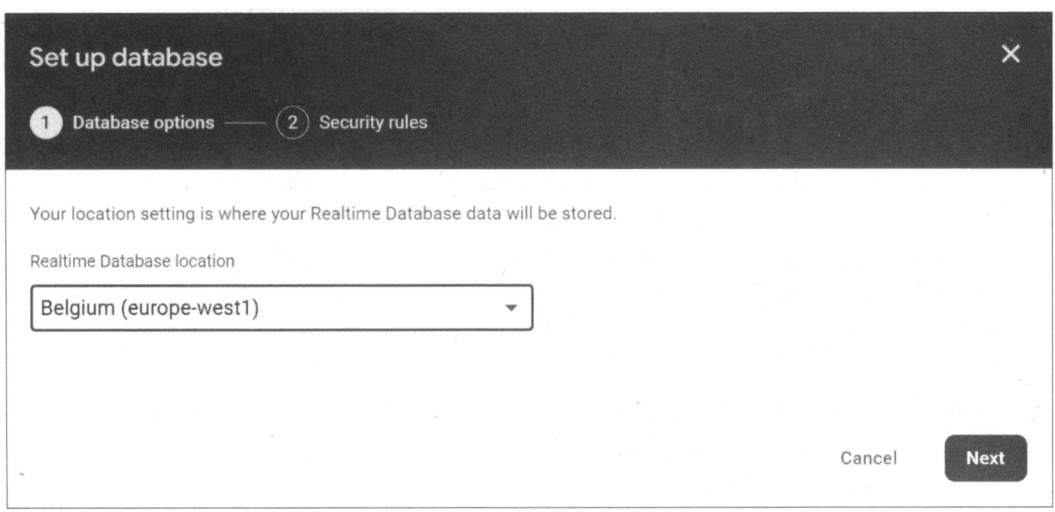


Рис. 2.4. Создание базы данных, страница 1

На заметку

Доступны для выбора также пункты **United States** (США) и **Singapore** (Сингапур).

19. Нажмем на кнопку **Next**, чтобы перейти ко второй странице диалогового окна.

    На второй странице (рис. 2.5) выбирается режим работы базы данных: закрытый или тестовый.

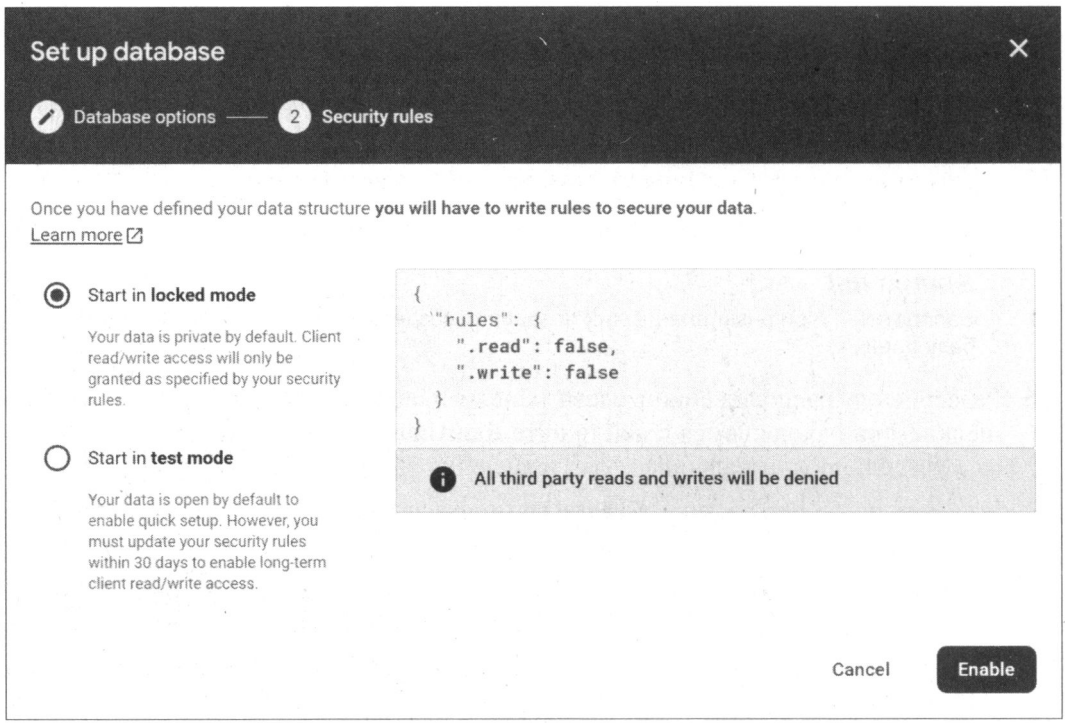


Рис. 2.5. Создание базы данных, страница 2

20. Оставим установленным переключатель **Start in locked mode**, задающий закрытый режим.

    База данных, работающая в закрытом режиме, по умолчанию недоступна ни для кого. Чтобы сделать ее доступной для всех или определенных посетителей, следует явно указать права доступа к ней (что мы скоро сделаем).

    Если же выбрать тестовый режим, база в течение 30 дней будет доступной для любых посетителей, что позволит протестировать работающее с ней приложение. Однако по прошествии 30 дней база переключится в закрытый режим, и, опять же, у нее придется указывать какие-либо права доступа.

21. Нажмем кнопку **Enable**, чтобы создать базу данных.

    Консоль выведет сведения о только что созданной базе данных, открытые на вкладке **Data** (рис. 2.6).

22. Найдем среди сведений о базе данных ее интернет-адрес и скопируем его.

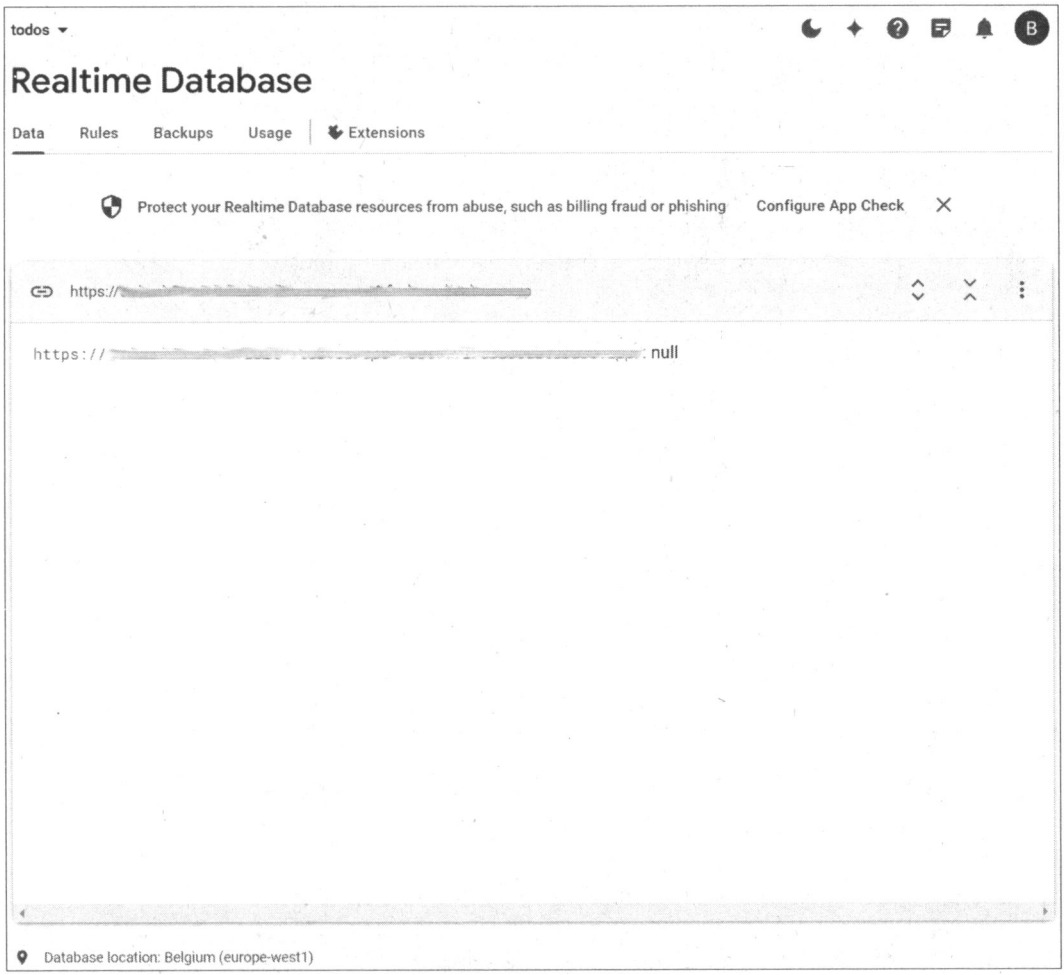


Рис. 2.6. Сведения о базе данных, вкладка **Data**

23. Добавим в объект с параметрами серверного приложения, полученный на *шаге 13*, свойство `databaseURL`, хранящее строку со скопированным интернет-адресом базы данных, внеся в код из модуля src\firebase.js следующие правки:

```
const firebaseConfig = {
    . . .
    appId: . . .,
    databaseURL: . . .
};
```

24. Переключимся на вкладку **Rules** сведений о базе данных (рис. 2.7).

    Ранее говорилось, что база данных, работающая в закрытом режиме, изначально недоступна для всех посетителей. Поэтому мы сейчас зададим ей такие права доступа, чтобы она стала доступна только для зарегистрированных посетителей.

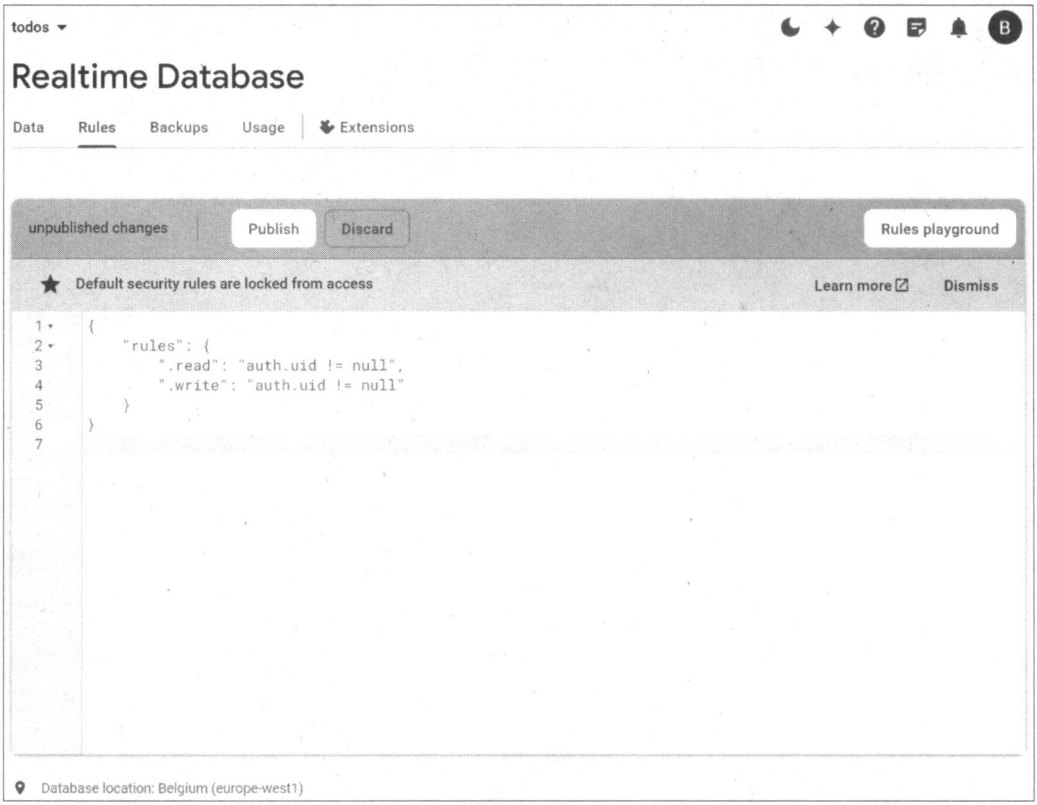


Рис. 2.7. Сведения о базе данных, вкладка **Rules**

25. Введем в большую область редактирования на вкладке **Rules** JSON-код из листинга 2.1.

Листинг 2.1. Код, дающий доступ к базе данных зарегистрированным посетителям

```
{
    "rules": {
        ".read": "auth.uid != null",
        ".write": "auth.uid != null"
    }
}
```

26. Нажмем на кнопку **Publish** над областью редактирования, чтобы сохранить («опубликовать») заданные права.

    Осталось добавить в проект подсистему разграничения доступа на основе адреса электронной почты и пароля.

27. Развернем в расположенной слева панели навигации группу пунктов **Build** и щелкнем на находящемся в ней пункте **Authentication**, чтобы переключиться в раздел настройки разграничения доступа.

28. Если откроется информационная страница — нажмем на кнопку **Getting started** для перехода непосредственно в раздел разграничения доступа.
29. Переключимся на вкладку **Sign-in method** раздела.

    Появится перечень провайдеров — подсистем Firebase, реализующих различные способы входа посетителей в базу данных (рис. 2.8).

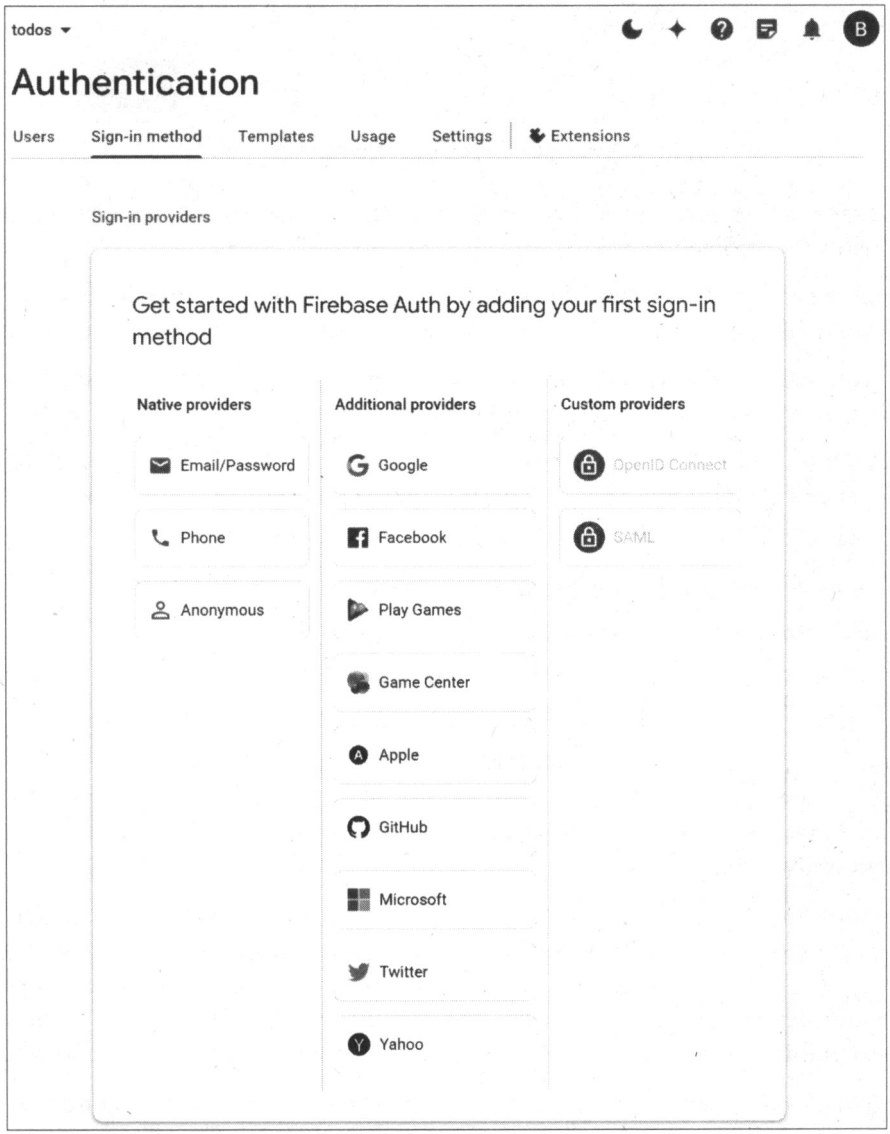


Рис. 2.8. Раздел настройки разграничения доступа, вкладка **Sign-in methods**

30. Найдем в колонке **Native providers** перечня провайдеров пункт **Email/Password**, представляющий провайдер, который реализует вход по адресу электронной почты и паролю, и щелкнем на этом пункте.

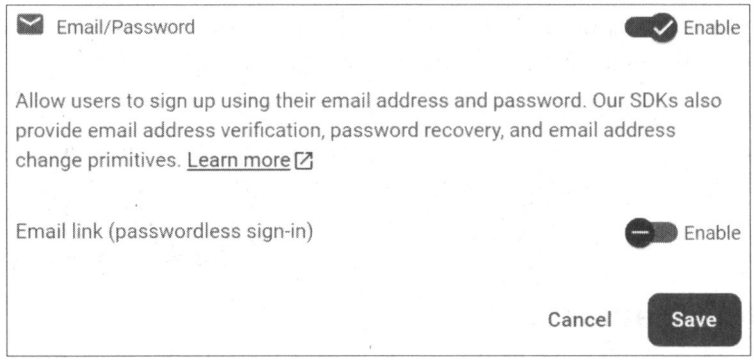


Рис. 2.9. Настройки провайдера **Email/Password**

Появится небольшое диалоговое окно для настройки этого провайдера (рис. 2.9). Сейчас мы активизируем его.

31. Установим в графе **Email/Password** этого диалогового окна переключатель **Enable** и нажмем на кнопку **Save**, чтобы сохранить измененные настройки.

    На этом манипуляции с консолью Google Firebase закончены. Остальные действия мы выполним непосредственно с проектом нашего React-приложения.

32. Установим клиентскую библиотеку Google Firebase SDK, предназначенную для работы со службой Firebase, для чего наберем в командной консоли команду:

    ```
    npm install firebase
    ```

33. Добавим в модуль src\firebase.js код для подключения к Firebase:

    ```
    import { initializeApp } from 'firebase/app';

    const firebaseConfig = {
        . . .
    };

    const firebaseApp = initializeApp(firebaseConfig);
    export default firebaseApp;
    ```

    Функция `initializeApp()` из модуля `firebase/app` выполняет подключение к серверному приложению Google Firebase. В качестве параметра она принимает простой объект с настройками подключения и возвращает объект, представляющий серверное приложение. Этот объект следует сохранить в какой-либо переменной и экспортировать из модуля, поскольку далее он нам понадобится.

Итак, серверный проект, серверное веб-приложение и база данных Firebase созданы, подсистема разграничения доступа настроена, а проект React-приложения подготовлен для работы с ними.

2.2. Регистрация новых посетителей

Чтобы зарегистрироваться в нашем приложении, посетитель перейдет на экран с веб-формой для ввода регистрационных сведений и занесет в нее свой адрес электронной почты и произвольный пароль. После нажатия кнопки регистрации эти сведения будут отправлены подсистеме разграничения доступа Google Firebase. Если новый посетитель был успешно создан, служба автоматически выполнит вход от его имени, в противном случае — пришлет код ошибки.

Для регистрации посетителя создадим компонент `Register` и ссылающийся на него маршрут с шаблонным путем **/register**. Регистрацию нового посетителя выполним в воздействии `register()`, которое запишем в этот маршрут.

Если вход в приложение не был выполнен, в меню навигации будет выведен пункт **Зарегистрироваться**, указывающий на экран регистрации. Если же вход был выполнен, в меню навигации появится пункт **Создать дело**, а в обозначении бренда вместо надписи «Todos» — адрес электронной почты *текущего посетителя* (посетителя, который в настоящий момент вошел в приложение).

Начнем с объявления воздействия `register()`, которое зарегистрирует в службе Firebase нового посетителя с заданными адресом электронной почты и паролем. Если регистрация прошла успешно, воздействие выполнит перенаправление на экран перечня дел, в противном случае — вернет строковый код ошибки. Это воздействие мы потом поместим в маршрут, указывающий на экран регистрации.

1. Добавим в модуль загрузчиков и воздействий src\api.js код воздействия `register()`:

```
. . .
import { getAuth, createUserWithEmailAndPassword } from 'firebase/auth';

import firebaseApp from './firebase.js';

const auth = getAuth(firebaseApp); // ..................................  1
. . .
export async function register({ request }) {
    const fd = await request.formData();
    try {
        const oUC = await createUserWithEmailAndPassword( // ............  2
            auth, // ..................................................  4
            fd.get('email'), fd.get('password') // ......................  3
        );
        return redirect('/'); // ...........................................  5
    }
    catch(err) {
        return err.code; // ................................................  6
    }
}
```

Сначала следует получить объект, представляющий подсистему разграничения доступа. Сделать это можно вызовом функции `getAuth()` из модуля `firebase/auth` с передачей ей объекта серверного приложения из модуля src\firebase.js (поз. 1 в представленном коде).

Функция `createUserWithEmailAndPassword()` из модуля `firebase/auth` (поз. 2) регистрирует в Firebase нового посетителя с адресом электронной почты и паролем, заданными вторым и третьим параметрами (поз. 3). Первым параметром указывается объект подсистемы разграничения доступа, полученный ранее (поз. 4).

В случае успеха производим перенаправление на экран перечня дел (поз. 5). Если же регистрация посетителя прошла неудачно, перехватываем возникшее исключение и возвращаем строковый код ошибки, извлеченный из свойства `code` объекта исключения (поз. 6).

После успешной регистрации посетителя служба Firebase автоматически выполнит вход от его имени.

2. Создадим модуль src\Register.js и запишем в него код компонента-экрана `Register` из листинга 2.2.

Листинг 2.2. Компонент регистрации посетителя `Register` (src\Register.js)

```
import { useState } from 'react';
import { useFetcher } from 'react-router-dom';

export default function Register(){
    const [email, setEmail] = useState('');
    const [password, setPassword] = useState('');
    const fetcher = useFetcher(); // ..................................  1

    const handleFormSubmit = evt => {
        evt.preventDefault();
        fetcher.submit({email, password}, // ..........................  2
                       {action: '/register', method: 'post'});
    };

    const handleFormReset = () => {
        setEmail('');
        setPassword('');
    };
    return (
        <section>
            <h1>Регистрация</h1>
            <form onSubmit={handleFormSubmit} onReset={handleFormReset}>
                <div className="field">
                    <label className="label">Адрес электронной почты</label>
```

```
                    <div className="control">
                        <input type="email" value={email} className="input"
                               onChange={e => setTitle(e.target.value)} />
                    </div>
                </div>
                <div className="field">
                    <label className="label">Пароль</label>
                    <div className="control">
                        <input type="password" value={password}
                             className="input"
                             onChange={e => setPassword(e.target.value)} />
                    </div>
                </div>
                    <div className="field is-grouped is-grouped-right">
                    <div className="control">
                        <input type="reset"
                               className="button is-warning is-light"
                               value="Сброс" />
                    </div>
                    <div className="control">
                        <input type="submit" className="button is-primary"
                               value="Зарегистрироваться" />
                    </div>
                </div>
            </form>
        </section>
    );
}
```

Элементы управления в веб-форме регистрации обрабатываются таким же образом, как и присутствующие в веб-форме создания нового дела (см. *упражнение 1.8*).

Когда посетитель нажмет кнопку отправки данных в веб-форме, будет предпринята попытка зарегистрировать нового посетителя в службе Google Firebase. Если адрес электронной почты и пароль нового посетителя введены и другого посетителя с таким адресом в службе еще нет, то регистрация пройдет успешно, и произойдет перенаправление.

Но если адрес электронной почты и пароль посетителя не введены или заданный адрес совпадает с адресом уже зарегистрированного в Google Firebase посетителя, при регистрации возникнет ошибка, и воздействие вернет код этой ошибки. В таком случае компоненту регистрации необходимо получить этот код, чтобы потом вывести сообщение об ошибке регистрации.

Таким образом, нам необходимо отправить из компонента регистрации воздействию `register()` клиентский запрос *без навигации на какой-либо другой компонент*.

Сделать это очень просто. Вызвав хук `useFetcher()` из модуля `react-router-dom` (поз. 1 в приведенном примере), мы получим объект *фетчера* — инструмента для отправки клиентских запросов без навигации. Фетчер поддерживает метод `submit()`, который выполняет ту же задачу, что и отправитель запросов, выданный хуком `useSubmit()`, и вызывается так же (поз. 2).

3. Добавим в список маршрутов src\routes.js маршрут, который укажет на экран регистрации, не забыв добавить в него только что написанное воздействие `register()`:

```
. . .
import Register from './Register.js';

~~import { getTodos, addTodo, getTodo, actTodo } from './api.js';~~
import { getTodos, addTodo, getTodo, actTodo, register } from './api.js';

const router = createBrowserRouter(
    createRoutesFromElements(
        <Route path='/' element={<App />}>
            . . .
            <Route path='register' element={<Register />}
                   action={register} />
        </Route>
    )
);
. . .
```

Осталось добавить в меню навигации гиперссылку на экран регистрации и сделать так, чтобы она выводилась лишь в том случае, если посетитель уже выполнил вход. И здесь мы столкнемся с проблемой...

Каждый компонент-экран в нашем приложении присутствует на странице в течение ограниченного времени — лишь до того момента, пока посетитель не переключится на другой экран. Следовательно, проверять, выполнил ли посетитель вход, мы можем однократно перед выводом экрана — в загрузчике, указанном в маршруте, который ведет на этот экран (займемся этим в *разд. 2.5*).

Но компонент приложения, в котором находится меню навигации, присутствует на экране постоянно. Значит, нам нужно как-то отслеживать изменение статуса посетителя — выполнил он вход, вошел в приложение или вышел из него. К счастью, библиотека Google Firebase SDK предоставляет такую возможность, позволяя зарегистрировать заданную функцию в качестве обработчика изменения статуса посетителя.

Напишем небольшую функцию `setStateChangeHandler(<функция>)`, которая зарегистрирует заданную *функцию* в качестве такого обработчика. Поместим ее объявление в модуле src\api.js.

4. Добавим в модуль src\api.js объявление этой функции:

```
. . .
import { getAuth, createUserWithEmailAndPassword } from 'firebase/auth';
import { getAuth, createUserWithEmailAndPassword,
         onAuthStateChanged } from 'firebase/auth';
. . .
export function setStateChangeHandler(func) {
    return onAuthStateChanged(auth, func);
}
```

Регистрацию обработчика изменения статуса посетителя производит функция `onAuthStateChanged()` из модуля `firebase/auth`. Первым параметром ей передается объект подсистемы разграничения доступа, вторым — сам регистрируемый обработчик.

Функция `onAuthStateChanged()` возвращает *функцию очистки*, которая отменяет регистрацию заданного обработчика. Эту функцию мы возвращаем в качестве результата из функции `setStateChangeHandler()` — она нам пригодится позже.

Функция-обработчик изменения статуса посетителя в качестве параметра получает объект, содержащий сведения о текущем посетителе, или `null`, если посетитель вышел из приложения. Этот объект сейчас понадобится, чтобы вывести в панели навигации адрес электронной почты текущего посетителя.

5. Добавим в компонент `App` необходимый код:

```
import { useState } from 'react';
import { useState, useEffect } from 'react';
. . .
import { setStateChangeHandler } from './api.js';
. . .
export default function App() {
    . . .
    const [user, setUser] = useState(); // ..............................  1
    . . .
    const authStateChanged = __user => { // ..............................  2
        setUser(__user);
    };

    useEffect(() => { // ..................................................  3
        const unsubscribe = setStateChangeHandler(authStateChanged);
        return () => {unsubscribe()}; // ..................................  4
    }, []); // ...........................................................  5

    return (
        <div className="container">
            <nav className="navbar is-light">
                <div className="navbar-brand">
                    <NavLink . . . >
                        Todos
```

```
                        {user ? user.email : 'Todos'} // ..............  6
                    </NavLink>
                    . . .
                </div>
                <div . . . >
                    <div className="navbar-start">
                        {user && ( // ..............................  7
                            <NavLink to="/add" . . . >
                                Создать дело
                            </NavLink>
                        )}
                        {!user && ( // ..............................  8
                            <NavLink to="/register"
                             className={({ isActive }) =>
                              'navbar-item' + (isActive ? ' is-active' : '')}
                            >
                                Зарегистрироваться
                            </NavLink>
                        )}
                    </div>
                </div>
            </nav>
        . . .
      </div>
    );
  }
}
```

Прежде всего, мы создали в компоненте новое состояние `user` (поз. 1 в приведенном коде). Оно будет хранить объект текущего посетителя, если таковой выполнил вход в приложение, или `null` — в противном случае.

Далее мы объявили вложенную функцию `authStateChange()` (поз. 2), которая будет обрабатывать изменение статуса посетителя и заносить полученное с параметром значение — объект посетителя или `null` — в состояние `user` компонента.

Эту функцию нужно зарегистрировать в качестве обработчика изменения статуса посетителя, передав ее ранее написанной функции `setStateChangeHandler()` из модуля src\api.js. Однако подобного рода код, обращающийся к внешним по отношению к React программным интерфейсам (в нашем случае к библиотеке Google Firebase SDK), не следует помещать непосредственно в код компонента — это может нарушить его работу. Как поступить?

Нам поможет хук `useEffect()` из модуля `react`. Он выполняет переданную ему в качестве первого параметра функцию строго в тот момент, когда код компонента уже выполнен, но сам компонент еще не выведен на экран (если это первоначальный вывод) или не обновлен (если компонент был выведен на экран

ранее и теперь обновляется). Функция, выполняемая в этот момент, носит название *эффекта*.

То, что нам нужно! Мы оформили регистрацию обработчика `authStateChange()` в виде функции-эффекта и передали ее хуку `useEffect()` (поз. 3).

Эффект может вернуть в качестве результата функцию очистки. Последняя будет вызвана перед каждым последующим обновлением и размонтированием (удалением со страницы) компонента и обычно применяется для отмены действий, выполненных в эффекте. Наша функция очистки вызывает функцию, выданную обработчиком `authStateChange()` и отменяющую его регистрацию (поз. 4).

По умолчанию эффект выполняется как при монтировании (первоначальном выводе) компонента, так и при каждом его обновлении. Однако обработчик достаточно зарегистрировать только один раз — при монтировании. Чтобы дать такое предписание React, мы указали вторым параметром в вызове хука `useEffect()` пустой массив (поз. 5).

В обозначении бренда выводим — в зависимости от статуса посетителя — его адрес электронной почты или надпись «Todos» (поз. 6), а в левой части меню навигации — пункт **Создать дело** (поз. 7) или **Зарегистрироваться** (поз. 8).

Щелкнем на пункте **Зарегистрироваться** в полосе навигации, перейдем на экран регистрации (рис. 2.10) и попробуем зарегистрироваться в приложении под произвольными адресом электронной почты (можно использовать и несуществующий) и паролем. После успешной регистрации и последующего автоматического входа мы увидим указанный нами почтовый адрес в левой части полосы навигации (рис. 2.11).

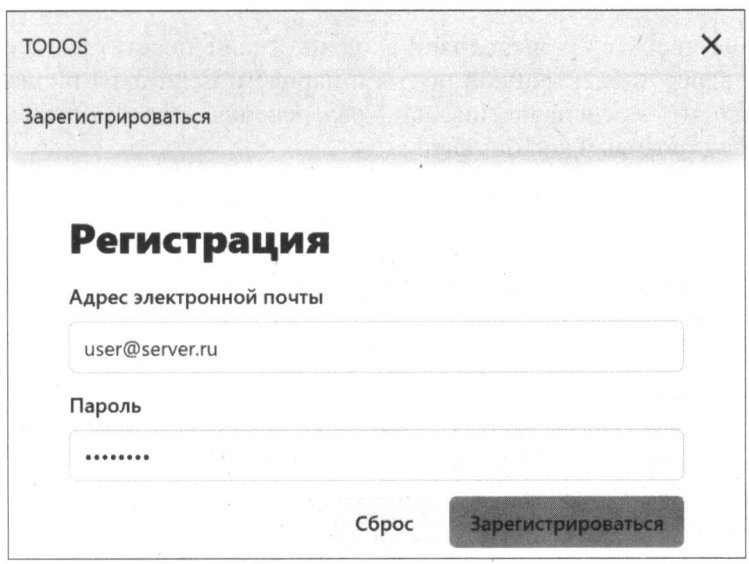


Рис. 2.10. Регистрация нового посетителя: меню навигации открыто, виден пункт **Зарегистрироваться**

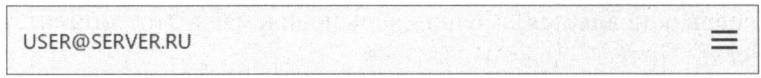


Рис. 2.11. Адрес электронной почты текущего посетителя в полосе навигации

На заметку

Вы можете просмотреть список всех посетителей, зарегистрированных в Google Firebase. Перейдите по интернет-адресу **https://firebase.google.com/**, откройте консоль, выберите вверху нужный проект, в панели навигации слева щелкните на пункте **Realtime Database** и переключитесь на вкладку **Users**.

За кулисами

Клиентская библиотека Google Firebase SDK после успешного входа сохраняет сведения о текущем посетителе на стороне клиента (в базе данных IndexedDB или локальном хранилище). Поэтому посетитель после входа в приложение может перейти на другой сайт или вообще закрыть веб-обозреватель — а после возврата в приложение ему не придется выполнять вход заново.

2.3. Вход и выход

Реализуем в нашем приложении вход и выход. Вход будет выполняться в компоненте-экране `Login`, на который поведет маршрут с шаблонным путем **/login**. Выход же организуем воздействием, которое запишем в маршруте с путем **/logout**, а отдельного компонента для выхода создавать не станем.

Поместим в левую часть меню навигации пункт **Войти**, доступный, если посетитель еще не выполнил вход, а в правую часть — пункт **Выйти**, доступный, напротив, после успешного входа.

Сначала объявим:

- воздействие `login()` — выполняет вход от имени посетителя с переданными в запросе адресом электронной почты и паролем. Если вход произошел успешно, производит перенаправление на экран перечня дел, в противном случае — возвращает строковый код ошибки;
- загрузчик `logout()` — выполняет выход и перенаправляет посетителя на экран входа.

1. Добавим в модуль src\api.js объявления воздействия `login()` и загрузчика `logout()`:

```
. . .
import { getAuth, createUserWithEmailAndPassword,
         onAuthStateChanged } from 'firebase/auth';
import { getAuth, createUserWithEmailAndPassword,
         onAuthStateChanged, signInWithEmailAndPassword,
         signOut } from 'firebase/auth';
. . .
export async function login({ request }) {
    const fd = await request.formData();
```

```
    try {
        await signInWithEmailAndPassword(
            auth, fd.get('email'), fd.get('password')
        );
        return redirect('/');
    }
    catch(err) {
        return err.code;
    }
}
export async function logout() {
    await signOut(auth);
    return redirect('/login');
}
```

Для выполнения входа следует вызвать функцию `signInWithEmailAndPassword()` из модуля `firebase/auth`, передав ей объект подсистемы разграничения доступа, адрес и пароль посетителя. А выход из приложения выполняется вызовом функции `signOut()` из того же модуля также с передачей ей объекта подсистемы разграничения доступа.

Компонент `Login` будет очень похож на компонент `Register` (см. *разд. 2.2*). Так что вы, уважаемые читатели, можете написать его самостоятельно.

2. Создайте самостоятельно компонент `Login` и сохраните его код в модуле src\Login.js.
3. Добавим в модуль src\routers.js код, создающий необходимые маршруты:

```
. . .
import Login from './Login.js';

~~import { getTodos, addTodo, getTodo, actTodo, register } from './api.js';~~
import { getTodos, addTodo, getTodo, actTodo, register, login, logout }
    from './api.js';

const router = createBrowserRouter(
    createRoutesFromElements(
        <Route path='/' element={<App />}>
            . . .
            <Route path='login' element={<Login />}
                   action={login} />
            <Route path='logout' loader={logout} />
        </Route>
    )
);
. . .
```

Во втором маршруте, ведущем на загрузчик `logout()`, который выполняет выход, мы не указали целевой компонент. Такое допускается при условии, если

загрузчик, записанный в маршруте, производит перенаправление (как в нашем случае).

4. Добавим в компонент App код, выводящий в меню навигации необходимые пункты:

```
export default function App() {
    . . .
    return (
        <div className="container">
            <nav className="navbar is-light">
                . . .
                <div . . . >
                    <div className="navbar-start">
                        {user && (
                            <NavLink . . . >
                                Создать дело
                            </NavLink>
                        )}
                        {!user && (
                            <NavLink to="/login"
                             className={({ isActive }) =>
                             'navbar-item' + (isActive ? ' is-active' : '')}
                            >
                                Войти
                            </NavLink>
                        )}
                        . . .
                    </div>
                    {user && (
                        <div className="navbar-end">
                            <NavLink to="/logout" className="navbar-item">
                                Выйти
                            </NavLink>
                        </div>
                    )}
                </div>
            </nav>
            . . .
        </div>
    );
}
```

В компонент NavLink, создающий гиперссылку **Выйти**, мы не вписывали код, помечающий ее как активную. В этом нет необходимости, потому что она никогда не станет активной, поскольку загрузчик, записанный в соответствующем маршруте, после выполнения сразу произведет перенаправление на другой экран.

Закончив с реализацией входа и выхода, начнем «учить» приложение работать с базой данных Google Firebase.

2.4. Хранение данных в базе Google Firebase

Теория

СУБД, являющаяся частью службы Firebase, является *документной*, т. к. хранит данные в *документах*, описывающих какие-либо отдельные сущности (например, дела) и представленных в виде простых объектов JavaScript. Каждый документ содержит произвольный набор *атрибутов* (аналогичны свойствам простых объектов, представляющих документы), которые могут хранить строки, числа, логические величины, `null`, другие документы или *коллекции* (аналоги массивов) из произвольного числа прочих значений, включая документы. В «основании» базы данных должен быть один документ, называемый *корневым*, — все остальные значения, документы и коллекции должны храниться в его свойствах.

Каждый документ, хранящийся в коллекции, должен иметь уникальный идентификатор. Он может как указываться явно при добавлении документа, так и присваиваться самой СУБД, — в последнем случае он представляет собой строку, сформированную на основе текущих даты и времени. Так что документы в любой коллекции уже выстроены по хронологическому признаку.

Вот пример небольшой базы данных, представленный в виде JSON-кода:

```
{"users" : { // ..................................................  1
  "9bBtqvTbByes7AIZTlWrviPoU1o1" : { // ..........................  2
    "todos" : { // ................................................  3
      "-Mfrhf9_Y76XrNvXhSo9" : { // ...............................  4
        "createdAt" : "18.07.2024, 18:10:24", // ...................  6
        "desc" : "Да поскорее!",
        "done" : true,
        "image" : "",
        "title" : "Изучить React"
      },
      "-Mfrht2wBLBmtza0zETW" : { // ...............................  5
        "createdAt" : "19.07.2024, 19:14:47",
        "desc" : "Список запланированных дел",
        "done" : false,
        "image" : "",
        "title" : "Написать первое React-приложение"
      }
    }
  },
  . . .
}}
```

Корневой документ имеет единственный атрибут `users` (поз. 1 в приведенном примере), хранящий коллекцию документов, каждый из которых представляет одного

посетителя приложения. Каждый документ в этой коллекции хранится под уникальным идентификатором (так, первый документ имеет идентификатор `9bBtqvTbByes7AIZTlWrviPoU1o1` — см. поз. 2) и содержит атрибут `todos` (поз. 3), в котором находится коллекция документов-дел. Отдельный документ-дело также имеет уникальный идентификатор (поз. 4 и 5) и набор атрибутов, хранящих элементарные значения: строки и логические величины (поз. 6).

Чтобы получить доступ к какой-либо коллекции, отдельному документу или отдельному атрибуту в документе, следует записать указывающий на него *путь*, представляющий набор имен атрибутов и идентификаторов, разделенные слешами. Так, чтобы получить коллекцию всех дел, созданных посетителем с идентификатором `9bBtqvTbByes7AIZTlWrviPoU1o1`, следует записать путь:

```
users/9bBtqvTbByes7AIZTlWrviPoU1o1/todos
```

Для получения дела с идентификатором `-Mfrht2wBLBmtza0zETW` — путь:

```
users/9bBtqvTbByes7AIZTlWrviPoU1o1/todos/-Mfrht2wBLBmtza0zETW
```

А для получения описания этого дела — путь:

```
users/9bBtqvTbByes7AIZTlWrviPoU1o1/todos/-Mfrht2wBLBmtza0zETW/desc
```

Практика

Реализуем в нашем приложении работу с базой данных Firebase: хранение в ней списка дел, считывание списка, добавление дела, пометку его как выполненного и удаление.

Начнем с добавления новых дел. Перепишем воздействие `addTodo()` таким образом, чтобы переданное ему новое дело записывалось в базу данных — в коллекцию по пути формата:

```
users/<идентификатор посетителя>/todos
```

Еще объявим служебную функцию `getUserId()`, которая будет возвращать идентификатор текущего посетителя или `undefined`, если вход в приложение не был выполнен.

1. Перепишем в модуле src\api.js код воздействия `addTodo()`:

```
. . .
import { getDatabase, ref, push, set } from 'firebase/database';
. . .
~~import todos from './todos.js';~~ // ..................................  1
. . .
const database = getDatabase(firebaseApp); // ..........................  2
. . .
function getUserId() {
    return auth.currentUser?.uid; // ....................................  3
}
. . .
```

```
export async function addTodo({ request }) {
    const currentUserId = getUserId(); // ..............................  4
    . . .
    const newTodo = {
        . . .
        key: date.getTime() // ..........................................  5
    };
    todos.push(newTodo);
    const db = ref(database, `users/${currentUserId}/todos`); // ........  6
    const r = await push(db); // ....................................... 7
    await set(r, newTodo); // ........................................... 8
    return redirect('/');
}
```

Поскольку мы отныне будем хранить все планируемые дела в базе данных, массив `todos` из модуля src\todos.js нам больше не нужен, и мы можем удалить и выражение импорта этого массива (поз. 1 в приведенном примере), и сам упомянутый модуль.

Для взаимодействия с базой данных понадобится объект, представляющий эту базу данных. Получить его можно вызовом функции `getDatabase()` из модуля `firebase/database` с передачей ей объекта серверного приложения из модуля src\firebase.js (поз. 2).

В функции `getUserId()` идентификатор текущего посетителя мы получаем из свойства `uid` объекта сведений о посетителе, который хранится в свойстве `currentUser` объекта подсистемы разграничения доступа (поз. 3).

В теле воздействия `addTodo()` получаем идентификатор текущего посетителя вызовом только что написанной функции `getUserId()` (поз. 4).

Ранее говорилось, что каждому документу при создании присваивается автоматически генерируемый уникальный идентификатор. Следовательно, нам самим генерировать этот идентификатор и сохранять его в свойстве `key` каждого дела больше не нужно, и мы можем удалить код, который это выполняет (поз. 5).

Сначала получаем ссылку на коллекцию, в которой будут храниться дела, для чего вызовем функцию `ref()`, передав ей первым параметром объект базы данных, полученный ранее, в вторым параметром — путь, по которому хранится коллекция дел (поз. 6). В передаваемый путь мы подставили идентификатор текущего посетителя, полученный ранее.

Готовый объект ссылки передаем асинхронной функции `push()`, которая создаст уникальный идентификатор для нового документа, добавит его к переданной ссылке, сформировав таким образом путь, по которому будет располагаться новый документ, и вернет новую ссылку, указывающую на этот путь (поз. 7).

Имея ссылку на новый, пока еще «пустой» документ, запишем в него новое дело, вызвав функцию `set()` с упомянутой ранее ссылкой и добавляемым делом в качестве параметров (поз. 8).

Все использованные здесь функции Google Firebase SDK объявлены в модуле `firebase/database`.

2. Исправим в модуле src\api.js код загрузчика `getTodos()` таким образом, чтобы он загружал список дел из базы данных:

```
. . .
import { getDatabase, ref, push, set } from 'firebase/database';
import { getDatabase, ref, push, set, set, query, get }
    from 'firebase/database';
. . .
export function getTodos() {
export async function getTodos(user) {
    return todos;
    const currentUserId = getUserId();
    const r = ref(database, `users/${currentUserId}/todos`); // ..... 1
    const q = query(r); // ..... 2
    const s = await get(q); // ..... 3
    const res = [];
    s.forEach((doc) => { // ..... 4
        const __todo = doc.val(); // ..... 5
        __todo.key = doc.key; // ..... 6
        res.push(__todo); // ..... 7
    });
    return res; // ..... 8
}
```

Как и в воздействии `addTodo()` (см. *шаг 1*), мы создаем ссылку на путь, по которому хранятся дела текущего посетителя (поз. 1 в приведенном примере). Эту ссылку вызовом функции `query()` преобразуем в запрос (поз. 2), который далее передаем функции `get()` (поз. 3). Последняя вернет *снимок* — объект, хранящий, в зависимости от указанной ранее ссылки, либо коллекцию документов (как в нашем случае), либо отдельный документ.

Функции `query()` и `get()` Google Firebase SDK объявлены в модуле `firebase/database`.

Перебираем документы-дела, содержащиеся в полученном снимке, с помощью метода `forEach()`, знакомого по массивам (поз. 4). Функции, указываемой в вызове этого метода, при каждом вызове передается снимок с очередным документом из коллекции.

В теле этой функции сначала извлекаем дело, представленное в виде простого объекта, из очередного снимка, вызвав у него метод `val()` (поз. 5). Полученный таким образом объект дела не содержит идентификатор документа (который нам нужен), поэтому мы сами извлекаем его из свойства `key` снимка и присваиваем свойству `key` дела (поз. 6). Готовое дело добавляем в созданный ранее массив (поз. 7).

Наконец, возвращаем готовый массив дел в качестве результата (поз. 8).

3. Перепишем в модуле src\api.js код загрузчика `getTodo()` для работы с базой данных:

```
~~export function getTodo({ params }) {~~
export async function getTodo({ params }) {
    ~~// . . . Старый код . . .~~
    const currentUserId = getUserId();
    const r = ref(database, `users/${currentUserId}/todos/${params.key}`);
    const q = query(r);
    const s = await get(q);
    if (!s.exists())
        throw new Error();
    return s.val();
}
```

Мы создаем ссылку на дело с заданным идентификатором, преобразуем эту ссылку в запрос и получаем снимок. Проверяем, существует ли запрошенное дело, вызвав у хранящего его снимка метод `exists()`, и, если не существует, возбуждаем исключение. В противном случае возвращаем дело, содержащееся в снимке.

4. Исправим в модуле src\api.js код воздействия `actTodo()` для работы с базой данных:

```
. . .
~~import { getDatabase, ref, push, set, query, get }~~
    ~~from 'firebase/database';~~
import { getDatabase, ref, push, set, query, get, remove }
    from 'firebase/database';
. . .
export function actTodo({ params, request }) {
    ~~// . . . Старый код . . .~~
    const currentUserId = getUserId();
    if (request.method === 'PATCH') {
        const r = ref(database,
                      `users/${currentUserId}/todos/${params.key}/done`);
        set(r, true);
    } else {
        const r = ref(database,
                      `users/${currentUserId}/todos/${params.key}`);
        remove(r);
    }
    return redirect('/');
}
```

Чтобы пометить дело с указанным идентификатором как выполненное, создаем ссылку на путь формата:

`users/<`*идентификатор посетителя*`>/todos/<`*идентификатор дела*`>/done`

где `done` в конце пути — это атрибут документа, хранящий признак завершения дела, и записываем в этот атрибут значение `true`, воспользовавшись функцией `set()`.

Удаление документа, находящегося по заданной ссылке, выполняет функция `remove()` из модуля `firebase/database`.

Однако после сохранения кода и обновления открытого в веб-обозревателе приложения мы получим сообщение об ошибке. React сообщит, что не может прочитать значение свойства `uid` у величины `null` — то есть извлечь идентификатор текущего посетителя, поскольку вход не был выполнен. Но мы-то ранее выполнили вход — когда регистрировались в приложении. Что случилось?

Google Firebase SDK по возможности сохраняет сведения о посетителе, выполнившем вход, во встроенной в веб-обозреватель базе данных IndexedDB. Но взаимодействие с этой базой данных производится асинхронно[1]. И данные из нее временами просто не успевают считаться к тому моменту, как в них возникает необходимость.

Когда приложение запускается, сначала выполняется код программного ядра самого React, ядер библиотек Google Firebase SDK и React Router. Google Firebase SDK, помимо всего прочего, запускает процесс считывания из базы IndexedDB сведений о текущем посетителе. Сразу же после этого React Router производит навигацию на экран перечня дел (путь /), в процессе которой выполняет загрузчик `getTodos()`. Последний пытается получить идентификатор текущего посетителя, который еще не считан из базы, и вполне предсказуемо «вываливается» по ошибке. И лишь после этого из базы загружаются данные текущего посетителя...

В качестве решения этой проблемы можно параллельно хранить данные посетителя в каком-либо другом хранилище, работа с которым выполняется синхронно, — например, в локальном хранилище Web API. Поскольку из всех этих данных нам требуется только идентификатор посетителя, мы можем сохранять там лишь его. Так и сделаем.

5. Исправим в коде модуля src\api.js воздействия `register()`, `login()`, `logout()` и служебную функцию `getUserId()`:

```
function getUserId() {
    return auth.currentUser?.uid;
    if (auth.currentUser)
        return auth.currentUser.uid;
    else
        return window.localStorage.getItem('user-id');
}
. . .
export async function register({ request }) {
    const fd = await request.formData();
```

[1] Надо отметить, что работа со встроенной базой данных IndexedDB реализована на редкость неудобно.

```
    try {
        await createUserWithEmailAndPassword(
        const cr = await createUserWithEmailAndPassword( // ..... 1
            auth,
            fd.get('email'), fd.get('password')
        );
        window.localStorage.setItem('user-id', cr.user.uid); // ..... 2
        return redirect('/');
    }
    . . .
}
. . .
export async function login({ request }) {
    const fd = await request.formData();
    try {
        await signInWithEmailAndPassword(
        const cr = await signInWithEmailAndPassword(// ..... 3
            auth, fd.get('email'), fd.get('password')
        );
        window.localStorage.setItem('user-id', cr.user.uid); // ..... 4
        return redirect('/');
    }
    . . .
}

export async function logout() {
    await signOut(auth);
    window.localStorage.removeItem('user-id'); // ..... 5
    return redirect('/login');
}
```

Функция `createUserWithEmailAndPassword()`, регистрирующая нового посетителя, возвращает объект с результатами регистрации и последующего входа (поз. 1 в приведенном коде). Свойство `user` этого объекта хранит объект со сведениями о посетителе, аналогичный содержащемуся в свойстве `currentUser` подсистемы разграничения доступа. Мы извлекаем из свойства `uid` объекта посетителя его идентификатор и сохраняем его в локальном хранилище под именем `user-id` (поз. 2).

Функция `signInWithEmailAndPassword()`, выполняющая вход, возвращает такой же объект, как и функция `createUserWithEmailAndPassword()` (поз. 3). Мы также извлекаем из него идентификатор посетителя и сохраняем в локальном хранилище (поз. 4).

При выходе из приложения удаляем ранее сохраненный идентификатор (поз. 5).

Перейдем по интернет-адресу **http://localhost:3000/login/** и выполним вход от имени ранее созданного посетителя. Попробуем добавить несколько новых дел, пометить какие-либо из них как сделанные и удалить. Далее выйдем из приложения, зарегистрируемся под другим адресом электронной почты и паролем, добавим от

имени вновь созданного посетителя еще несколько дел и также выйдем из приложения.

> ***На заметку***
>
> Вы можете просмотреть содержимое базы данных, созданной в Google Firebase. Перейдите по интернет-адресу **https://firebase.google.com/**, откройте консоль, выберите вверху нужный проект, в панели навигации слева щелкните на пункте **Realtime Database** и переключитесь на вкладку **Data**. Структура базы данных представляется в виде иерархического списка.

> ***За кулисами***
>
> Данные, загруженные из базы Firebase, кешируются для ускорения доступа к ним на стороне клиента (в базе IndexedDB или локальном хранилище). Все манипуляции с этими данными (добавление, правка и удаление документов) сначала выполняются с их локальной копией, а уже потом в фоновом режиме — с базой Firebase, что повышает производительность.

2.5. Разграничение доступа

Сделаем так, чтобы посетитель смог получить доступ к экранам перечня дел, сведений о выбранном деле и создания нового дела только после успешного входа. Если же на эти экраны попытается попасть *гость* (посетитель, не выполнивший вход), будем выполнять перенаправление его на экран входа.

Выяснить, был ли выполнен вход, очень просто. Нужно лишь в каждом загрузчике, загружающем закрытые от гостей данные, и в каждом воздействии, изменяющем закрытые данные, проверять, доступен ли идентификатор текущего посетителя.

1. Исправим в модуле src\api.js код загрузчиков `getTodos()` и `getTodo()`:

```
export async function getTodos() {
    const currentUserId = getUserId();
    if (!currentUserId)
        return redirect('/login');
    . . .
}
. . .
export async function getTodo({ params }) {
    const currentUserId = getUserId();
    if (!currentUserId)
        return redirect('/login');
    . . .
}
```

   Если идентификатор текущего посетителя не доступен, значит, вход еще не был выполнен, и нужно намекнуть об этом посетителю.

2. Самостоятельно исправим в модуле src\api.js код воздействий `addTodo()` и `actTodo()` аналогичным образом.

Еще в *разд. 2.2* и *2.3* мы условились сделать экраны регистрации и входа доступными лишь для гостей. И сделали так, чтобы гиперссылки, ведущие на эти экраны, показывались лишь гостям.

Однако любой посетитель все же сможет попасть на любой из этих экранов, набрав интернет-адрес **http://localhost:3000/register/** или **http://localhost:3000/login/** соответственно. Это проблема, и ее нужно решить.

Напишем загрузчик `onlyLoggedOut()`, в коде которого станем проверять, выполнил ли посетитель вход, и, если выполнил, производить перенаправление на экран перечня дел.

3. Добавим в модуль src\api.js объявление этого загрузчика:

```
export function onlyLoggedOut() {
    if (getUserId())
        return redirect('/');
    else
        return null;
}
```

4. Исправим код модуля src\routes.js, добавив загрузчик `onlyLoggedOut()` в маршруты, ведущие на экраны регистрации и входа:

```
. . .
~~import { getTodos, addTodo, getTodo, actTodo, register, login, logout }~~
    ~~from './api.js';~~
import { getTodos, addTodo, getTodo, actTodo, register, login, logout,
         onlyLoggedOut } from './api.js';

const router = createBrowserRouter(
    createRoutesFromElements(
        <Route path='/' element={<App />}>
            . . .
            <Route path='register' element={<Register />}
                   ~~action={register} />~~
                   action={register} loader={onlyLoggedOut} />
            <Route path='login' element={<Login />}
                   ~~action={login} />~~
                   action={login} loader={onlyLoggedOut} />
            . . .
        </Route>
    )
);
. . .
```

Выполним выход из приложения и попытаемся попасть на экран создания нового дела, перейдя по интернет-адресу **http://localhost:3000/add**. Если мы вместо этого попадем на экран входа, значит, разграничение доступа работает. После чего войдем в приложение под именем какого-либо из зарегистрированных посетителей, попытаемся попасть на экран входа и убедимся, что нас перенаправили на перечень дел.

2.6. Валидация и вывод сообщений об ошибках в стиле React

Теория

Для *валидации* (проверки данных, занесенных в веб-форму, на корректность согласно заданным правилам) и вывода сообщений об ошибках ввода можно использовать средства, предоставляемые языком HTML (например, атрибут тега `required`):

```
<label>Заголовок</label>: <input . . . required />
```

Однако в React-приложениях удобнее реализовать программную валидацию, которая позволяет, во-первых, выполнять проверку данных по правилам более сложным, нежели реализуемые средствами HTML, а во-вторых, выводить более исчерпывающие сообщения об ошибках ввода.

Программная валидация в стиле React реализуется в два этапа:

1. Для каждого элемента управления — в компоненте создается состояние, хранящее сообщение об ошибке ввода, которое относится к этому элементу, или пустую строку, если значение, занесенное в элемент, корректно.
2. Перед обработкой введенных данных (например, перед отправкой их бэкенду) — занесенное в элемент значение проверяется на корректность, и в созданное ранее состояние заносится сообщение об ошибке ввода или пустая строка, если введенное значение корректно.

Если введенные данные отправляются бэкенду, аналогично можно реализовать вывод сообщений об ошибках, выданных бэкендом.

Практика

Реализуем в компоненте регистрации валидацию данных согласно следующим правилам:

- адрес электронной почты — обязателен к занесению;
- пароль — обязателен к занесению;
- повторно введенный пароль (для задания которого мы создадим еще одно поле ввода) — должен совпадать с ранее введенным паролем.

Сообщения об ошибках ввода, равно как и сообщения об ошибках, выданных Google Firebase, станем выводить под соответствующими полями ввода.

Начнем с подготовительных действий:

- поместим в веб-форму еще одно поле ввода — для повторного занесения пароля;
- создадим состояние `passwordConfirm` — для хранения повтора пароля;
- добавим обработчик события `change` поля повтора пароля — для переноса повтора пароля в состояние `passwordConfirm`;

- создадим состояния `errorEmail`, `errorPassword` и `errorPasswordConfirm` — для хранения сообщений об ошибках ввода соответственно адреса электронной почты, пароля и повтора пароля;
- реализуем вывод сообщений об ошибках под соответствующими полями ввода.

1. Добавим необходимый код в компонент `Register`:

```
export default function Register() {
    . . .
    const [passwordConfirm, setPasswordConfirm] = useState('');
    const [errorEmail, setErrorEmail] = useState('');
    const [errorPassword, setErrorPassword] = useState('');
    const [errorPasswordConfirm, setErrorPasswordConfirm] = useState('');
    . . .
    const handleFormReset = () => {
        setEmail('');
        setPassword('');
        setPasswordConfirm('');
    };
    . . .
    return (
        <section>
            <h1>Регистрация</h1>
            <form . . .>
                <div className="field">
                    <label className="label">Адрес электронной почты</label>
                    <div className="control">
                        . . .
                    </div>
                    {errorEmail &&
                        <p className="help is-danger">
                            {errorEmail}
                        </p>
                    }
                </div>
                <div className="field">
                    <label className="label">Пароль</label>
                    <div className="control">
                        . . .
                    </div>
                    {errorPassword &&
                        <p className="help is-danger">
                            {errorPassword}
                        </p>
                    }
                </div>
```

```
                <div className="field">
                    <label className="label">Повтор пароля</label>
                    <div className="control">
                        <input type="password" value={passwordConfirm}
                          className="input"
                          onChange={e => setPasswordConfirm(e.target.value)}
                        />
                    </div>
                    {errorPasswordConfirm &&
                        <p className="help is-danger">
                            {errorPasswordConfirm}
                        </p>
                    }
                </div>
                . . .
            </form>
        </section>
    );
}
```

Для вывода сообщений об ошибках поместим под каждый блок со стилевым классом `control`, внутри которого помещается поле ввода, абзац со стилевыми классами `help` (стандартное оформление подобного рода предупреждений в стиле Bulma — в частности, уменьшенный кегль шрифта) и `is-danger` (красный цвет текста). Этот абзац будет выводиться лишь в том случае, если указано соответствующее сообщение об ошибке.

Для выполнения валидации объявим вложенную функцию `validate()`, которая вернет `true`, если занесенные данные корректны, и `false` — в противном случае. А чтобы удалить со страницы все выведенные ранее сообщения об ошибках, дополнительно объявим служебную вложенную функцию `resetErrorMessages()`.

2. Добавим в код компонента `Register` объявления вложенных функций `resetErrorMessages()` и `validate()`:

```
export default function Register() {
    . . .
    const resetErrorMessages = () => {
        setErrorEmail('');
        setErrorPassword('');
        setErrorPasswordConfirm('');
    };

    const validate = () => {
        resetErrorMessages(); // ..................................... 1
        if (!email) { // ............................................. 2
            setErrorEmail('Адрес электронной почты не указан'); // ..... 3
            return false; // ........................................... 4
        }
```

```
        if (!password) {
            setErrorPassword('Пароль не указан');
            return false;
        }
        if (!passwordConfirm) {
            setErrorPasswordConfirm('Повтор пароля не указан');
            return false;
        }
        if (password !== passwordConfirm) {
            setErrorPassword('Введенные пароли не совпадают');
            setErrorPasswordConfirm('Введенные пароли не совпадают');
            return false;
        }
        return true; // ..... 5
    };
    . . .
    return ( . . . );
}
```

В теле функции `validate()` сначала сбрасываем сообщения об ошибках вызовом функции `resetErrorMessages()` (поз. 1 в приведенном коде). Далее проверяем первое из введенных значений на некорректность (поз. 2). Если оно некорректно, заносим сообщение об ошибке ввода в соответствующее состояние (поз. 3) и завершаем выполнение функции с возвратом значения `false` (поз. 4). Если первое значение корректно, подвергаем валидации второе значение и т. д. В случае корректности всех значений возвращаем `true` (поз. 5).

3. Добавим в тело компонента `Register` код для вывода сообщений об ошибках, возвращенных Google Firebase:

```
export default function Register() {
    . . .
    const resetErrorMessages = () => { // ..... 2
        . . .
    };

    if (fetcher.data) { // ..... 3
        resetErrorMessages(); // ..... 1
        if (fetcher.data === 'auth/email-already-in-use') { // ..... 4
            setErrorEmail('Посетитель с таким адресом ' + // ..... 5
                          'электронной почты уже зарегистрирован');
        } else if (fetcher.data === 'auth/weak-password') {
            setErrorPassword('Слишком простой пароль');
            setErrorPasswordConfirm('Слишком простой пароль');
        }
        fetcher.data = undefined; // ..... 6
    }
    . . .
}
```

Поскольку в этом коде выполняется вызов вложенной функции `resetErrorMessages()` (поз. 1 в приведенном примере), мы поместили код после объявления этой функции (поз. 2).

Ответ, полученный от вызванного воздействия (в нашем случае код ошибки, выданный воздействием `register()`), можно получить из свойства `data` фетчера (поз. 3, фетчеры описывались в *разд. 2.2*).

Если от воздействия был получен код ошибки (поз. 3), мы удаляем все сообщения об ошибках (поз. 1), выясняем, какой именно код ошибки был получен (поз. 4), и выводим соответствующее сообщение (поз. 5). Напоследок очищаем данные, полученные от воздействия, чтобы избежать зацикливания компонента с последующим крахом всего приложения (поз. 6).

Осталось добавить вызов функции `validate()` в нужное место функции `handleFormSubmit()`, обрабатывающей щелчок на кнопке регистрации.

4. Исправим в компоненте `Register` код функции `handleFormSubmit()`:

```
export default default Register() {
    . . .
    const handleFormSubmit = evt => {
        evt.preventDefault();
        if (validate())
            fetcher.submit({email, password},
                           {action: '/register', method: 'post'});
    };
  . . .
}
```

Вызываем воздействие, регистрирующее нового посетителя, только если функция `validate()` вернет `true`.

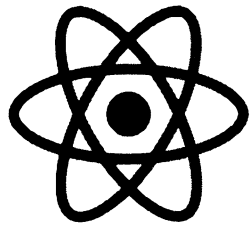
Выйдем из приложения. Зарегистрируем еще одного, третьего по счету, посетителя с произвольными адресом электронной почты и паролем. Попытаемся зарегистрироваться снова — с пустыми адресом электронной почты, паролем, повтором пароля, с несовпадающими паролями, с адресом, уже зарегистрированным в Firebase, и, наконец, с заведомо простым паролем (например, 123). Проверим, выводятся ли под соответствующими полями ввода сообщения об ошибках.

На этом разработка учебного веб-приложения списка планируемых дел закончена.

> ***Готовое веб-приложение...***
>
> ...хранится в папке todos-2-0 сопровождающего книгу электронного архива (см. *приложение 3*).

В качестве домашнего задания реализуйте валидацию подобного рода в компонентах входа и добавления нового дела. Коды ошибок, выдаваемых функцией `signInWithEmailAndPassword()`, которая выполняет вход, вы найдете на странице:

https://firebase.google.com/docs/reference/js/v8/firebase.auth.Auth?hl=en#signinwithemailandpassword.



ЧАСТЬ II

Базовые инструменты

Глава 3. Проект React-приложения

Глава 4. Компоненты

Глава 5. Язык JSX

Глава 6. Обработка событий

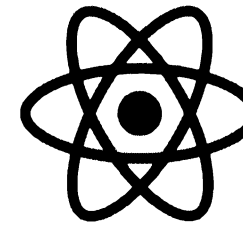
Глава 7. Взаимодействие между компонентами

Глава 8. Работа с веб-формами и элементами управления

Глава 9. Работа с внешними API

Глава 10. Компоненты высшего порядка и собственные хуки

ГЛАВА 3



Проект React-приложения

https://t.me/it_boooks/2

Проект — это совокупность всех файлов, составляющих приложение: модулей на языке JavaScript, таблиц стилей, веб-страницы приложения, графических изображений, файлов с данными в формате JSON, аудио- и видеофайлов и др.

Все содержимое проекта должно находиться в одной папке — *папке проекта*. Она может иметь произвольное местоположение в файловой системе и произвольное имя. Папку проекта можно переименовать и (или) переместить в другое место — это не скажется на работоспособности проекта.

3.1. Создание React-проекта

Для создания нового React-проекта следует в командной консоли перейти в папку, в которой должна находиться папка создаваемого проекта, и подать команду формата:

```
npx create-react-app[@<версия>] <имя проекта> [--template <имя шаблона>]
```

Если *версия* не указана, будет задействована наиболее актуальная версия утилиты create-react-app из существующих.

Пример:

```
npx create-react-app todos
```

А следующая команда демонстрирует создание проекта с использованием утилиты create-react-app версии 5.0.1 — актуальной на момент подготовки этой книги:

```
npx create-react-app@5.0.1 todos
```

Если заданная версия утилиты create-react-app еще ни разу не запускалась, утилита npx выведет предупреждение о необходимости загрузить ее из репозитория Node.js и установить на компьютер (после названия утилиты через символ @ показан номер ее версии):

```
Need to install the following packages:
create-react-app@5.0.1
Ok to proceed? (y)
```

На это предупреждение следует ответить положительно, последовательно нажав клавиши <Y> (от *англ.* yes) и <Enter>.

В результате будет создан проект в папке с указанным *именем*.

Командный ключ `--template` задает *имя шаблона*, на основе которого будет создан проект. *Шаблон* React-приложения служит своего рода заготовкой для создания проекта, содержит указания на установку необходимых зависимостей (библиотек, включая сам React, и инструментальных программ) и начальный программный код (в частности, стартовый модуль и компонент приложения).

Если ключ `--template` не указан, проект будет создан на основе базового шаблона `cra-template`. Он предусматривает лишь необходимый минимум библиотек и минималистичный компонент приложения, показанный на рис. 1.1.

Другие доступные для указания шаблоны можно найти в репозитории Node.js[1]. Они содержат всевозможные дополнительные библиотеки (React Router, Redux, Bootstrap и др.) и программный код, реализующий какие-либо типовые операции с использованием этих библиотек.

> ***Внимание!***
> Далее будет описываться работа с проектом, созданным на основе базового шаблона `cra-template`.

3.2. Структура React-проекта

В папке вновь созданного проекта изначально присутствуют следующие папки и файлы:

- public — папка, предназначенная для хранения файлов, которые не подвергаются никакой обработке перед отправкой веб-обозревателю. Изначально в ней находятся следующие файлы:
  - index.html — единственная веб-страница, на который выводится интерфейс приложения (*страница приложения*). Будет рассмотрена позже;
  - favicon.ico — значок приложения. Представляет собой логотип React;
  - robots.txt — файл с указаниями для поисковых служб;
  - manifest.json — манифест прогрессивного веб-приложения (PWA). Требуется при программировании такого веб-приложения;
  - logo192.png — значок PWA размерами 192×192 пиксела;
  - logo512.png — значок PWA размерами 512×512 пикселов.

  Если разрабатываемое приложение не планируется превращать в PWA, файлы manifest.json, logo192.png и logo512.png могут быть удалены. При этом необ-

[1] См. **https://www.npmjs.com/search?q=cra-template-***.

ходимо также удалить из HTML-кода страницы приложения index.html ссылки на файлы manifest.json и logo192.png.

При запуске проекта на выполнение папка public выступает в качестве корневой папки веб-приложения. Все пути к файлам, присутствующие в интернет-адресах, отсчитываются от нее;

- src — папка для хранения файлов, которые обрабатываются перед отправкой веб-обозревателю. Изначально содержит файлы:
  - index.js — стартовый модуль;
  - index.css — таблица стилей приложения. Импортируется в модуле index.js;
  - App.js — модуль с кодом компонента приложения `App`;
  - App.css — таблица стилей компонента приложения. Импортируется в модуле App.js;
  - logo.svg — логотип React, выводимый компонентом приложения. Импортируется в модуле App.js;
  - App.test.js — тестовый модуль для компонента приложения. Если средства автоматизированного тестирования, устанавливаемые в составе инструментальных программ, использовать не планируется, может быть удален.

    Внимание!

    Автор не будет описывать встроенные в React средства автоматизированного тестирования, поскольку не считает их сколь-нибудь полезными.

  - reportWebVitals.js — модуль встроенного в React измерителя производительности приложения.

    Если измеритель производительности не нужен, его можно удалить. При этом также необходимо удалить из модуля index.js выражение, вызывающее функцию `reportWebVitals()`, которая импортируется из этого модуля, а также выражение, импортирующее эту функцию;

  - setupTests.js — модуль, инициализирующий средства автоматизированного тестирования React. Если эти средства использоваться не будут, модуль может быть удален;
- node_modules — папка с установленными зависимостями (библиотеками, включая сам React, и инструментальными программами);
- .gitignore — список исключений для системы управления версиями Git. Включает, в частности, папки node_modules, build и файлы окружения с именами формата .env[.<*окружение*>].local;
- package.json — манифест проекта Node.js;
- package-lock.json — снимок дерева зависимостей (или реестр) проекта Node.js;
- README.MD — краткое описание проекта в формате Markdown.

Кроме того, если на компьютере установлена система Git, в папке проекта создается Git-репозиторий.

Разработчик приложения может:

- создавать любые нужные для работы файлы в папках public и src;
- создавать в папках public и src любые вложенные папки для хранения файлов;
- исправлять любые файлы в папках public и src (например, файл с инструкциями для поисковых служб, манифест PWA, а также саму страницу приложения);
- заменять любые файлы в папках public и src.

  В частности, следует заменить изначальный значок страницы приложения public\favicon.ico на другой, хранящий логотип разрабатываемого приложения;
- удалять ненужные файлы.

  Например, файл src\logo.svg с изображением, которое выводится изначальным компонентом приложения, после переписывания последнего, скорее всего, окажется ненужным и может быть удален. Аналогично, если для оформления приложения применяется какой-либо CSS-фреймворк (наподобие Bootstrap или Bulma), можно удалить ненужные таблицы стилей src\index.css и src\App.css.

Ранее говорилось, что файлы с исходным кодом, находящиеся в составе React-проекта, хранятся в двух папках:

- src — содержащиеся в ней файлы подвергаются обработке: трансляции и объединению в связки (что это такое, кратко описывалось в *разд. 1.3*). При этом некоторые файлы могут быть переименованы, если пакет Webpack «посчитает», что без этого не обойтись.

  В этой папке следует размещать файлы, содержимое и работоспособность которых не пострадают в результате обработки, а именно:

  - JavaScript-модули;
  - внешние таблицы стилей, импортируемые в JavaScript-модулях (в качестве примера можно привести таблицы стилей src\index.css и src\App.css);
  - графические изображения, аудио- и видеоролики, шрифты, архивы и прочие файлы, также импортируемые в JavaScript-модулях (например, src\logo.svg);
- public — хранящиеся в ней файлы *не* обрабатываются и, в частности, не переименовываются.

  В этой папке следует размещать файлы, которые:

  - должны иметь строго определенные имена (например, страница приложения index.html, значок страницы favicon.ico, инструкции для поисковых служб robots.txt и др.). Переименование таких файлов приведет к нарушению работы приложения и даже полной его неработоспособности;
  - файлы, ссылки на которые записаны в HTML-коде страницы приложения и манифесте PWA: сам манифест PWA, значки (наподобие logo192.png и logo512.png), внешние таблицы стилей, внешние веб-сценарии и др. Переименование какого-либо из таких файлов вызовет «разрыв» ссылки и нарушение работы приложения.

3.3. Базовые файлы проекта

Базовых файлов, с которых все начинается, в React-проекте всего три: страница приложения, стартовый модуль и таблица стилей приложения.

3.3.1. Веб-страница приложения

Веб-страница приложения public\index.html открывается веб-обозревателем при открытии приложения и служит основой для вывода его интерфейса. Как правило, страница приложения содержит лишь минимальный набор метатегов и пустой блок (тег `<div>`) с якорем `root`, в котором выводится компонент приложения.

В листинге 3.1 показан HTML-код стартовой страницы вновь созданного React-проекта (присутствовавшие в коде объемистые комментарии удалены ради компактности).

Листинг 3.1. Веб-страница приложения (public\index.html)

```
<!DOCTYPE html> <!-- ..........................................  1 -->
<html lang="en">
  <head>
    <meta charset="utf-8" /> <!--.............................  2 -->
    <link rel="icon" href="%PUBLIC_URL%/favicon.ico" /> <!--.....  3 -->
    <meta name="viewport"
          content="width=device-width, initial-scale=1" /> <!-- ..  4 -->
    <meta name="theme-color" content="#000000" /> <!-- ..........  5 -->
    <meta
      name="description"
      content="Web site created using create-react-app"
    />
    <link rel="apple-touch-icon"
          href="%PUBLIC_URL%/logo192.png" /> <!-- ..............  6 -->
    <link rel="manifest" href="%PUBLIC_URL%/manifest.json" /> <!--  7 -->
    <title>React App</title>
  </head>
  <body>
    <noscript>You need to enable JavaScript to run this app.</noscript>
    <div id="root"></div> <!-- ..............................  8 -->
  </body>
</html>
```

В коде страницы присутствуют следующие примечательные элементы:

- пролог, указывающий на язык HTML 5, на котором написана страница, — поз. 1 в листинге 3.1;
- указание на использованную кодировку UTF-8 — поз. 2;

- ссылка на значок приложения — поз. 3.

  В коде приложения используются *переменные окружения*, которые хранят всевозможные данные, необходимые инструментальным программам из состава React-проекта. Например, переменная окружения `PUBLIC_URL`, присутствующая в интернет-адресе файла значка, хранит интернет-адрес корневой папки приложения (по умолчанию: /);

- указание установить ширину области просмотра равной ширине экрана и не масштабировать страницу — поз. 4.

  Это необходимо для того, чтобы при просмотре на мобильных устройствах масштаб страницы не уменьшался, и для нормальной работы CSS-фреймворков наподобие Bulma или Bootstrap;

- указание отобразить интерфейс веб-обозревателя черным цветом (поддерживается только мобильной редакцией Apple Safari) — поз. 5;

- ссылка на значок, которая будет использоваться для вывода веб-клипов, ссылающихся на создаваемое приложение (поддерживаются только Apple iOS), — поз. 6;

- ссылка на манифест PWA — поз. 7;

- пустой блок (тег `<div>`) с якорем `root`, в котором будет выведен компонент приложения, — поз. 8.

Разработчик при необходимости может:

- править содержимое страницы приложения. В частности, рекомендуется сразу же заменить ее изначальное название `React App`, заданное в теге `<title>`, на более конкретное (скажем, `Todos`).

  Можно заменить тег для вывода компонента приложения на другой (например, `<main>` со стилевым классом `container`, создающим контейнер Bulma):

```
<body>
  . . .
  <main id="root" class="container"></main>
</body>
```

- удалять ненужные метатеги. Так, если приложение не планируется превращать в PWA, можно удалить тег `<link>` со ссылкой на файл манифеста;

- добавлять произвольные метатеги с дополнительной информацией о странице.

Также допускается привязывать к странице файлы с внешними таблицами стилей и веб-сценариями в случаях, если:

- эти файлы загружаются с других сайтов;

- код, хранящийся в этих файлах, может потерять работоспособность, будучи обработанным инструментальными программами React-проекта. Впрочем, подобного рода ситуация встречается крайне редко.

3.3.2. Стартовый модуль

Стартовый модуль src\index.js загружается веб-обозревателем сразу после открытия страницы приложения (см. *разд. 3.3.1*), инициализирует и запускает приложение.

Листинг 3.2 показывает код стартового модуля (присутствовавшие в нем комментарии удалены).

Листинг 3.2. Стартовый модуль (src\index.js)

```
import React from 'react';
import ReactDOM from 'react-dom/client';
import './index.css'; // ..................................................  6
import App from './App';
import reportWebVitals from './reportWebVitals';

const root = ReactDOM.createRoot(document.getElementById('root')); // ......  1
root.render( // ..........................................................  2
    <React.StrictMode> // ................................................  4
        <App /> // ..........................................................  3
    </React.StrictMode>
);

reportWebVitals(); // ....................................................  5
```

Сначала React должен получить доступ к элементу, в котором следует вывести компонент приложения, — изначально это блок с якорем `root` (см. листинг 3.1). Для этого применяется знакомый всем веб-программистам метод `getElementById()`. На основе найденного элемента создается объект, способный выводить React-компоненты, для чего применяется функция `createRoot()` из модуля `react-dom/client` (поз. 1 в листинге 3.2).

Вывод какого-либо содержимого на страницу выполняет метод `render()` полученного ранее объекта (поз. 2). В параметре этой функции указывается выводимое содержимое, представленное JSX-кодом.

> ***На заметку***
>
> Функция `render()` из модуля `react-dom`, которая применялась для вывода содержимого ранее, в React 18 была объявлена устаревшей и не рекомендованной к применению, а в React 19 — удалена.

Выводимое содержимое представляет собой компонент приложения `App` (поз. 3). Он «обернут» в компонент `StrictMode` из модуля `react` (поз. 4), который активизирует строгий режим, способный помочь в выявлении ряда ошибок на раннем этапе разработки (подробности приведены в *главе 23*).

В самом конце вызывается функция `reportWebVitals()` из модуля src\reportWebVitals.js (поз. 5). Она подготавливает к выводу отчет о производительности приложения

(подробности — в *главе 23*). Если анализ производительности приложения выполнять не планируется, вызов этой функции можно закомментировать или удалить (в таком случае также нужно удалить выражение, импортирующее эту функцию, и модуль src\reportWebVitals.js, в котором она объявлена).

В стартовом модуле импортируется таблица стилей приложения src\index.css (поз. 6), которая задает оформление всего приложения.

Разработчик при необходимости может:

- изменить выводимое содержимое. Например, при использовании библиотеки React Router в блоке `root` вместо компонента приложения потребуется вывести провайдер маршрутизации;
- добавить выражения импорта таблиц стилей, поставляемых в составе используемого CSS-фреймворка;
- добавить любой другой требуемый код.

3.3.3. Таблица стилей приложения

Таблица стилей приложения src\index.css задает оформление, общее для всего приложения и не привязанное к какому-либо из компонентов. Обычно в ней определяются параметры фона страницы приложения и общее оформление различных тегов.

Если приложение имеет достаточно простое оформление, его можно полностью поместить в таблицу стилей приложения. В таком случае таблицы стилей отдельных компонентов (они описаны в *разд. 5.3.1*) окажутся ненужными и могут быть удалены.

Если для оформления приложения будет использован какой-либо CSS-фреймворк, таблица стилей приложения может оказаться не нужной. Тогда следует удалить как сам файл src\index.css, так и выражение, присутствующее в коде стартового модуля и выполняющее его импорт.

3.3.3.1. Сброс стилей

Ряд параметров оформления страниц (в частности, кегли шрифтов, межстрочный интервал, внешние и внутренние просветы у разных элементов) не имеют значений по умолчанию, явно прописанных в интернет-стандартах, вследствие чего у разных веб-обозревателей значения этих параметров различаются. Это приводит к тому, что одна и та же страница, открытая в разных веб-обозревателях, выглядит по-разному.

Поэтому перед разработкой веб-приложения рекомендуется *явно* указать у этих параметров какие-либо значения. Иначе говоря, выполнить *сброс стилей*.

Чтобы сбросить стили, достаточно вставить в начало таблицы стилей приложения следующую строку:

```
@import-normalize;
```

При запуске проекта на выполнение инструментальные программы React-проекта самостоятельно встроят в это место необходимые CSS-стили.

> ***За кулисами***
>
> Для выполнения сброса React применяет популярную CSS-библиотеку normalize.css[1].

Если для оформления приложения применяется какой-либо CSS-фреймворк, явно выполнять сброс стилей не нужно — фреймворк сделает это сам.

3.3.3.2. Использование таблиц стилей SASS

React позволяет использовать для оформления таблицы стилей, написанные на языке SASS. Для этого достаточно установить программу sass, набрав в командной консоли команду:

```
npm install sass
```

После чего можно создать таблицу стилей с расширением scss или sass и импортировать ее в стартовом модуле:

```
import './index.scss';
```

Таким же образом можно использовать таблицы стилей SASS, входящие в состав каких-либо фреймворков (например, Bulma):

```
import 'bulma/bulma.scss';
```

3.4. Запуск и остановка выполнения проекта

Чтобы с целью отладки запустить проект на выполнение (тем самым заставив приложение работать), в командной консоли следует перейти непосредственно в папку проекта и отдать команду:

```
npm [run] start
```

Слово `run` можно набрать, а можно и не набирать — оно ни на что не влияет.

> ***Внимание!***
>
> Запущенный таким образом проект будет работать в режиме разработки (development), наиболее подходящем для отладки приложения. При этом переменная окружения `NODE_ENV`, поддерживаемая самой Node.js, будет хранить значение `'development'`.

При подготовке проекта к запуску будут задействованы следующие инструментальные программы, входящие в состав проекта:

- *линтер* (утилита для проверки программного кода) ESLint — проверит код на предмет недочетов (например, объявленных и неиспользуемых переменных), случаев использования устаревших программных инструментов и синтаксических ошибок;

---

[1] См. **https://github.com/csstools/normalize.css**.

- транслятор Babel — транслирует JSX-код в обычный JavaScript, а языковые конструкции, появившиеся в последних версиях JavaScript и поддерживаемые еще не всеми используемыми в настоящее время веб-обозревателями, — в более старые и общеупотребительные;
- сборщик Webpack — собирает все файлы с программным кодом и таблицы стилей в единую связку.

> ***За кулисами***
>
> Связка сохраняется в нескольких файлах (*чанках*[1]), один из которых содержит библиотечный JavaScript-код, другой — код, управляющий загрузкой связки и добавляемый самим Webpack, остальные — JavaScript-код, написанный разработчиком приложения. Аналогично в разных чанках сохраняется CSS-код таблиц стилей. Разбиение связки на чанки позволяет оптимизировать их загрузку и кеширование на стороне клиента (скажем, библиотечный код изменяется редко, поэтому при следующем запуске приложения хранящий его чанк, скорее всего, не устареет и соответственно может быть извлечен из кеша, что ускорит запуск приложения).
>
> Все чанки при запуске в режиме разработки сохраняются в оперативной памяти — для ускорения сборки и предотвращения «замусоривания» диска файлами с устаревшим кодом. Также в память загружается копия страницы приложения, к которой при помощи тегов `<script>` привязываются все чанки.
>
> Эта страница и открывается веб-обозревателем на последней стадии запуска проекта.

- отладочный веб-сервер — запустится, станет работать через TCP-порт 3000 и обслуживать папку public в качестве корневой;
- веб-обозреватель, указанный в системных настройках как используемый по умолчанию, — автоматически запустится и выполнит переход по интернет-адресу **http://localhost:3000/**.

  Поскольку в качестве корневой служит папка public, при переходе по этому интернет-адресу веб-обозревателем будет загружена и открыта страница приложения public\index.html (точнее, ее копия, ранее сохраненная в памяти). В результате веб-приложение запустится.

В процессе выполнения указанных действий в консоли появятся соответствующие сообщения. В случае успешного запуска последним таким сообщением будет:

```
Compiled successfully!
```

выведенное зеленым шрифтом.

Если в программном коде встретится ошибка, в консоли будет выведено оповещение, и процесс запуска проекта прервется.

Чтобы остановить работу проекта, следует выполнить следующие шаги:

1. Переключиться в экземпляр консоли, в котором был запущен проект.
2. Нажать комбинацию клавиш <Ctrl>+<C> и подтвердить завершение работы, нажав клавиши <Y> и <Enter>.

   Затем можно просто закрыть этот экземпляр консоли, если он больше не нужен.

---

[1] От *англ.* chunk — порция, фрагмент.

После завершения работы проекта автоматически запущенный веб-обозреватель закрыт не будет.

3.4.1. Отслеживание кода

В процессе запуска проекта после отладочного веб-сервера будет запущена еще одна инструментальная программа — *монитор кода*. Она начнет отслеживать изменение состава проекта (изменение входящих в него файлов, добавление новых и удаление существующих), повторно транслировать и собирать изменившийся код, после чего отправлять веб-обозревателю указание загрузить его новую редакцию. Этот процесс называется *отслеживанием кода*.

В результате запущенное приложение станет обновляться автоматически, без необходимости в принудительном обновлении нажатием клавиши <F5>. А у компонентов будет сохранено состояние.

Автор книги рекомендует в процессе программирования вообще не останавливать работу проекта. У такого подхода есть два преимущества. Во-первых, как говорилось ранее, не придется самостоятельно обновлять работающее приложение. Во-вторых, запуск проекта может отнять довольно много времени — порядка минуты, что весьма существенно.

Тем не менее иногда (хоть и редко) механизм отслеживания кода дает сбои: измененный код некорректно считывается, транслируется, собирается или отправляется веб-обозревателю. В таких случаях приложение работает неправильно или не работает вообще, несмотря на отсутствие ошибок в коде. Если такое произошло, следует:

- обновить открытое в веб-обозревателе приложение принудительно — нажатием клавиши <F5>;
- если и это не помогло — завершить выполнение проекта и запустить его снова.

3.4.2. Использование HTTPS при отладке

Если разрабатываемое приложение работает с бэкендом, который разрешает взаимодействие только по протоколу HTTPS, следует указать отладочному веб-серверу, чтобы он также работал по этому протоколу. Для этого достаточно набрать в консоли следующую команду:

- в командной строке — `set HTTPS=true&&npm start`;
- в PowerShell — `($env:HTTPS = "true") -and (npm start)`.

Следует учесть, что React использует самоподписанный сертификат, поэтому веб-обозреватель перед открытием приложения выведет предупреждение о недопустимом сертификате, которое придется проигнорировать.

3.5. Выявление ошибок

3.5.1. Ошибки в программном коде

Ошибки в программном коде выявляются линтером и транслятором. Сообщения о таких ошибках выводятся непосредственно в экземпляре консоли, в котором был запущен проект.

- *Критическая ошибка* (например, синтаксическая) приведет к остановке процесса трансляции и прерыванию выполнения приложения. При этом в консоли красным шрифтом будет выведена строка:

  ```
  Failed to compile.
  ```

  Под ней появится непосредственно описание ошибки, написанное согласно формату:

  ```
  <путь к модулю, в котором присутствует ошибка>
    Line <№ строки>:<№ символа>:  <описание ошибки>
  ```

  Путь к модулю указан относительно папки проекта.

  Это сообщение будет выведено в двух экземплярах: первый — непосредственно линтером, второй — транслятором (который просто повторно выведет сообщение, выданное линтером)[1].

  Пример сообщения о непонятной языковой конструкции `retrn` (вероятно, вместо нее должно быть `return`):

  ```
  src\App.js
    Line 24:5:  'retrn' is not defined  no-undef
  ```

  Такое же сообщение (причем в одном экземпляре) появится и в веб-обозревателе, в котором ранее было запущено приложение.

- *Некритическая ошибка* не приведет к прерыванию трансляции кода, сборки проекта и выполнения приложения. Однако будет выведена коричневым шрифтом строка:

  ```
  Compiled with warnings.
  ```

  Далее появится, также в двух экземплярах, сообщение об ошибке приведенного ранее формата.

  Пример сообщения об объявленной, но не использованной переменной `val`:

  ```
  src\App.js
    Line 9:11:  'val' is assigned a value but never used  no-unused-vars
  ```

  В веб-обозревателе оно дублироваться не будет.

---

[1] На взгляд автора, это избыточно.

3.5.2. Ошибки времени выполнения

Ошибки времени выполнения — наиболее трудновыявляемые, поскольку вызываются неблагоприятным стечением обстоятельств, возникших при работе приложения.

Сообщения об ошибках времени выполнения выводятся непосредственно в веб-обозревателе. К сожалению, они малоинформативны и не позволяют сразу установить место в программном коде, в котором возникла ошибка[1]. Пример сообщения о подобной ошибке показан на рис. 3.1.

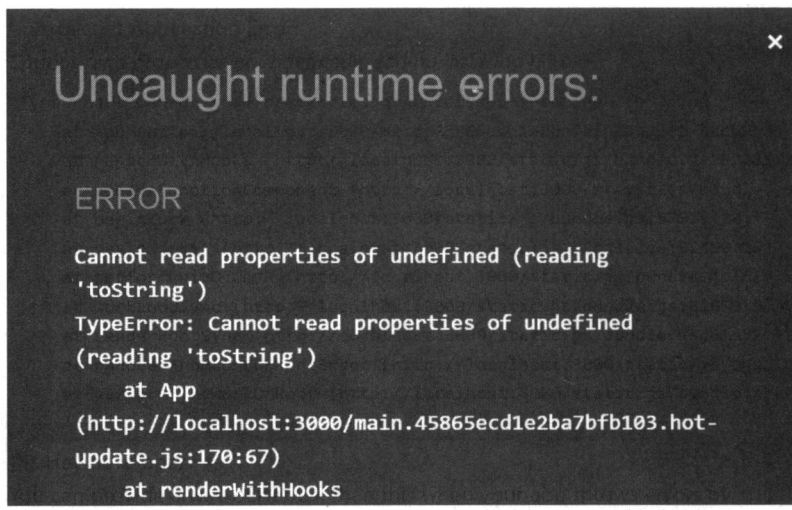


Рис. 3.1. Сообщение об ошибке времени выполнения (показано частично)

Наибольший интерес представляет верхняя часть сообщения, где описан характер возникшей ошибки (в нашем случае попытка вызвать метод `toString()` у значения `undefined`).

Сообщение об ошибке можно скрыть, щелкнув на кнопке закрытия, расположенной в правом верхнем углу сообщения и имеющей вид крестика. При этом в веб-обозревателе появится аналогичное сообщение, только еще менее удобное.

3.6. Настройка проекта. Переменные окружения

Проект React-приложения поддерживает ряд настроек. В частности, можно указать программу веб-обозревателя для запуска приложения и номер TCP-порта, через который будет работать отладочный веб-сервер.

Для задания настроек проекта служат *переменные окружения*. Они аналогичны переменным из языков программирования, только используются инструментальными

[1] В React 17 сообщения об ошибках были намного более информативными. Остается лишь надеяться, что в будущих версиях фреймворка устранят этот недочет.

программами из состава React-проекта: линтером, транслятором, сборщиком и веб-сервером.

Переменные окружения также применяются для хранения каких-либо значений, которые следует вставить в программный код приложения во время его трансляции и сборки. Например, можно записать название страницы в переменную окружения, а в HTML-тег `<title>`, находящийся в коде страницы, вставить ссылку на эту переменную, — и в процессе сборки инструментальные программы поместят в это место название, ранее сохраненное в переменной окружения.

3.6.1. Задание переменных окружения. Файлы окружения

Задать переменные окружения для проекта можно двумя способами:

- в специально предназначенных для этого файлах (*файлах окружения*) непосредственно в папке проекта. Обычно таким образом указываются переменные, чьи значения в дальнейшем, скорее всего, изменяться не будут.

  Поддерживаются следующие файлы окружения:

  - .env — общий файл окружения. В нем задаются настройки, используемые во всех случаях и всеми разработчиками, занимающимися проектом;
  - .env.local — *локальный файл окружения*. В отличие от общего, он не заносится в коммиты Git (в файле .gitignore есть соответствующее исключение).

    В локальном файле окружения хранятся настройки, заданные текущим разработчиком проекта сообразно его предпочтениям;
  - .env.development и .env.production — общие файлы окружения, задающие настройки для режима соответственно разработки и эксплуатации проекта;
  - .env.development.local и .env.production.local — локальные файлы окружения, задающие настройки для режима соответственно разработки и эксплуатации проекта.

  Файлы окружения указаны здесь в порядке возрастания их приоритета. Настройки, заданные в файлах с более высоким приоритетом, перекрывают настройки из менее приоритетных файлов. Так, если какая-либо настройка была указана в файлах .env и .env.local, будет использовано значение, записанное в файле .env.local.

  Содержимое файла окружения записывается в текстовом формате в виде набора строк, каждая из которых представляет одну настройку:

  <имя настройки>=<значение настройки>

  Например:

```
BROWSER=chrome
PORT=8000
```

  В файлы окружения можно помещать комментарии, предварив их двойным слешем:

```
// Приложение будет запущено в Google Chrome
BROWSER=chrome
```

> **Совет**
>
> Если вы пользуетесь редактором Visual Studio Code, установите расширение ENV. Оно выполняет синтаксическую подсветку содержимого файлов окружения и позволяет форматировать их.

- в консоли при запуске проекта — в формате:
  - в командной строке:

    ```
    set "<имя настройки>=<значение настройки>" && npm start
    ```

  - в PowerShell:

    ```
    ($env:<имя настройки> = <значение настройки>) -and (npm start)
    ```

  Указанная таким образом настройка будет действовать только пока проект запущен.

Если настройка нигде не указана, она получит значение по умолчанию, записанное в коде инструментальных программ React-проекта.

3.6.2. Использование переменных окружения в программном коде

Сослаться на какую-либо переменную окружения в коде приложения можно так:

- в HTML-коде — используя синтаксис следующего формата:

  ```
  %<имя переменной окружения>%
  ```

  Пример обращения к переменной окружения `PUBLIC_URL`:

  ```
  <link . . . href="%PUBLIC_URL%/manifest.json" />
  ```

- в JavaScript- и JSX-коде — через одноименное свойство объекта `process.env`. Примеры обращения к переменной окружения `BUILD_PATH`:

  ```
  const s = 'Файлы эксплуатационной редакции приложения хранятся ' +
            'в папке ' +  process.env.BUILD_PATH;
  . . .
  <p>Файлы эксплуатационной редакции приложения хранятся в папке
     {process.env.BUILD_PATH}</p>
  ```

3.6.3. Переменные окружения, поддерживаемые React

React поддерживает следующие переменные окружения:

- используемые только при разработке:
  - `BROWSER` — веб-обозреватель, служащий для запуска приложения: `chrome`, `firefox` или `edge`. Также можно указать полный путь к исполняемому файлу

нужного веб-обозревателя. По умолчанию задействуется веб-обозреватель, указанный в системных настройках как используемый по умолчанию;

- `BROWSER_ARGS` — параметры консоли, передаваемые веб-обозревателю при запуске (по умолчанию не передается никаких параметров);
- `HOST` — адрес хоста, на котором работает отладочный веб-сервер. По умолчанию отладочный сервер будет работать на всех доступных локальных адресах: **localhost**, **127.0.0.1** и IP-адресе текущего компьютера в локальной сети;
- `PORT` — номер TCP-порта, через который будет работать отладочный веб-сервер (по умолчанию: 3000).

  Две последние настройки может понадобиться изменить — например, если на текущем компьютере требуется запустить одновременно несколько React-проектов. В таком случае у разных проектов следует задать разные адреса хоста и (или) номера TCP-портов;
- `HTTPS` — если `true`, отладочный веб-сервер будет работать по протоколу HTTPS, если `false` — по протоколу HTTP (по умолчанию: `false`);
- `WDS_SOCKET_HOST` — адрес хоста, на котором будет работать сервер WebSocket, посредством которого веб-обозревателю в процессе отслеживания кода (см. *разд. 3.4.1*) будут посылаться указания загрузить файл с исправленным кодом (по умолчанию — совпадает с адресом хоста, указанным в переменной `HOST`, или **localhost**, если она не задана);
- `WDS_SOCKET_PORT` — номер TCP-порта, через который будет работать упомянутый ранее сервер WebSocket (по умолчанию — совпадает с номером порта, используемым отладочным веб-сервером);
- `WDS_SOCKET_PATH` — путь, на котором будет работать упомянутый ранее сервер WebSocket (по умолчанию: **/ws**, в версиях фреймворка, предшествовавших React 18, — **/sockjs-node**).

  Эти три последние настройки может потребоваться изменить — например, если на текущем компьютере требуется запустить одновременно несколько React-проектов, чтобы они при выполнении не «мешали» друг другу;
- `FAST_REFRESH` — если `true`, будет задействован поддерживаемый React механизм быстрого обновления компонентов, при использовании которого по возможности будет сохраняться их состояние. Если `false`, механизм быстрого обновления задействован не будет. По умолчанию: `true`.

  Этой настройке следует дать значение `false`, если открытое в веб-обозревателе приложение почему-то не обновляется при исправлении программного кода;
- `CHOKIDAR_USEPOLLING` — если `true`, монитор кода будет работать в режиме опроса (polling mode), если `false` — в обычном режиме (по умолчанию: `false`).

  Давать этой настройке значение `true` следует, если монитор почему-то не отслеживает изменение программного кода. Обычно такое случается при запуске проекта в виртуальной машине;

- ESLINT_NO_DEV_ERRORS — если true, синтаксические ошибки не будут приводить к прерыванию трансляции кода и сборки проекта (хотя приложение, в конце концов, так и не заработает). Если false, любая встреченная в коде синтаксическая ошибка прервет трансляцию кода. По умолчанию: false;

♦ используемые только при построении эксплуатационной редакции приложения:

- BUILD_PATH — путь к папке, в которой будут сохранены файлы эксплуатационной редакции. Можно указать как абсолютный путь, например:

  ```
  BUILD_PATH=d:/work/dist
  ```

  так и относительно папки проекта, предварив путь точкой:

  ```
  BUILD_PATH=./dist
  ```

  Значение по умолчанию: ./build;

- INLINE_RUNTIME_CHUNK — если true, программный код, запускающий загрузку связки, будет помещен непосредственно в страницу приложения public\index.html в качестве внутреннего веб-сценария. Если false, этот код будет вынесен во внешний веб-сценарий в отдельный JavaScript-файл. По умолчанию: true.

  Настройке следует дать значение false, если планируемая политика безопасности приложения запрещает выполнение внутренних веб-сценариев;

- GENERATE_SOURCEMAP — если true, помимо JavaScript-файлов с программным кодом будут сгенерированы соответствующие им файлы с картами исходного кода (source map), необходимые для отладки и трассировки. Если false, файлы с картами исходного кода генерироваться не будут. По умолчанию: true.

  Генерирование файлов с картами отнимает много системных ресурсов и на маломощных компьютерах может вызвать аварийный останов в связи с нехваткой памяти. В таких случаях следует задать у этой настройки значение false;

♦ используемые в обоих случаях:

- PUBLIC_URL — интернет-адрес приложения. Можно указать как путь, абсолютный или относительный, так и полный интернет-адрес. По умолчанию: /.

  Эта настройка задается, если приложение планируется хранить не в корневой папке веб-сервера, а во вложенной в нее папке или на другом веб-сервере (скажем, CDN). В первом случае в настройке записывается путь, например:

  ```
  PUBLIC_URL=/app
  ```

  во-втором — полный интернет-адрес, например:

  ```
  PUBLIC_URL=http://cdn.mysite.ru/static
  ```

- IMAGE_INLINE_SIZE_LIMIT — максимальный размер графического изображения, сохраняемого в виде отдельного файла, в байтах. Любое изображение меньше указанного размера будет преобразовано в data URL и вставлено не-

посредственно в CSS- или JavaScript-код. Если указать `0`, любые изображения будут сохраняться в отдельных файлах. По умолчанию: 10 000;

- `CI` — если `true`, при выдаче линтером, транслятором или сборщиком предупреждения о некритической ошибке (подробности — в *разд. 3.5.1*) создание эксплуатационной редакции приложения будет прервано. Если `false`, в таком случае приложение будет успешно собрано. По умолчанию: `false`.

  Задавать для этой настройки значение `true` следует при использовании каких-либо инструментальных программ для непрерывной интеграции;

- `DISABLE_EXLINT_PLUGIN` — если `true`, линтер ESLint будет отключен, если `false` — включен (по умолчанию: `false`);

- `TSC_COMPILE_ON_ERROR` — принимается во внимание, если программный код пишется на языке TypeScript. Если `true`, сборка проекта будет успешной даже в случае присутствия в исходном коде ошибок несоответствия типов. Если `false`, сборка проекта будет прервана. По умолчанию: `false`.

3.6.4. Создание своих переменных окружения

Допускается создание своих переменных окружения. Единственное требование: имена создаваемых переменных должны начинаться с префикса `REACT_APP_` — во избежание совпадения с именами уже существующих переменных окружения.

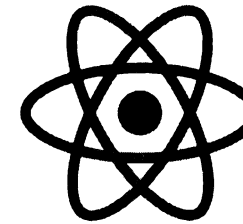
> ***Внимание!***
>
> **Все переменные окружения, имена которых не начинаются с префикса `REACT_APP`, за исключением `NODE_ENV` и описанных в *разд. 3.6.3*, будут игнорироваться инструментальными программами React-проекта.**

Пример создания и использования переменной окружения `REACT_APP_PAGE_TITLE`:

```
// Файл окружения .env
REACT_APP_PAGE_TITLE=Todos

<!-- Файл страницы приложения public\index.html -->
<html lang="en">
  <head>
    . . .
    <title>%REACT_APP_PAGE_TITLE%</title>
  </head>
  . . .
</html>
```

ГЛАВА 4



Компоненты

https://t.me/it_boooks/2

Компонент — это независимая часть интерфейса веб-приложения, часто рассчитанная на повторное использование. Компонент может иметь любое содержимое — от отдельного элемента страницы (наподобие кнопки) до целого экрана приложения (*компонент-экран*) — и быть сколь угодно сложным.

Компонент приложения выводится на страницу сразу после запуска приложения и вмещает в себя весь его интерфейс.

4.1. Реализация компонентов

Компоненты в React реализуются в виде обычных функций JavaScript.

> ***На заметку***
>
> Также поддерживается реализация компонентов в виде классов. Однако, начиная с React 18, компоненты-классы были объявлены устаревшими и не рекомендуются к применению. Они рассматриваются в *приложении 1*.

Функция, реализующая компонент, должна:

- иметь имя, начинающееся с большой буквы (это обязательно!). Примеры:

  `App`, `Register`, `Login`

  Если имя компонента состоит из нескольких слов, каждое из них должно начинаться с большой буквы, например:

  `TodoList`, `TodoDetail`, `TodoAdd`

  Такой стиль именования сущностей в программном коде называется *PascalCase*;

- принимать единственный параметр — простой объект с пропами, полученными компонентом от родителя.

  Если компонент не должен принимать пропы (как, например, компонент приложения), объявлять параметр необязательно;

- возвращать в качестве результата объектную структуру, представляющую содержимое компонента, которое должно быть выведено на странице. Эта объектная структура описывается с применением языка JSX.

В теле компонента можно объявлять любые локальные переменные и вложенные функции, необходимые для реализации его логики.

Пример простейшего компонента `TodoItem`, который выводит дело, переданное ему в качестве пропа (о них будет рассказано чуть позже), с оформлением в стиле Bulma, показан в листинге 4.1.

Листинг 4.1. Компонент дела `TodoItem`

```
export default function TodoItem(props) {
    return (
        <div className="box">
            <h2>{props.todo.title}</h2>
            <p>{props.todo.desc}</p>
        </div>
    );
}
```

Использовать этот компонент можно, например, следующим образом:

```
import TodoItem from './TodoItem.js';
. . .
export default function TodoList(props) {
    . . .
    return (
        <section>
            <h1>Дела</h1>
            {list.map(item => (
                <TodoItem todo={item} />
            ))}
        </section>
    );
}
```

Как правило объявление каждого компонента сохраняется в отдельном модуле и подвергается экспорту по умолчанию. Однако объявления мелких компонентов можно поместить в один модуль и подвергнуть множественному экспорту. Если компонент используется лишь в том модуле, в котором он объявлен, экспортировать его не нужно.

4.1.1. Жизненный цикл компонента. Хуки

Жизненный цикл (время существования на странице) компонента делится на три этапа:

- *монтирование* — первое отображение на странице сразу после создания;
- *обновление* — повторное отображение выведенного ранее компонента. Производится после изменения какого-либо из значений, относящихся к реактивным данным (пропов, состояний и др.). Выполняется быстрее, чем монтирование;

- *размонтирование* — удаление компонента со страницы. Производится при программном удалении компонента, переходе на другой экран в процессе навигации, переходе на другую веб-страницу или закрытии веб-обозревателя.

Есть возможность предписать компоненту выполнение заданных действий на любом из этих этапов: создать (при монтировании) и удалить (при размонтировании) состояние, исполнить указанную функцию (в качестве эффекта — при монтировании и обновлении) и др. Отдать компоненту такое предписание можно вызовом в его коде одной из специфических функций, называемых *хуками*.

Хуки представляют собой обычные функции JavaScript. По существующим соглашениям имя хука должно предваряться префиксом `use`.

> ***Внимание!***
>
> Хуки следует вызывать лишь непосредственно в теле компонента. Помещать вызовы хуков в условные выражения, циклы и тела вложенных функций не допускается и приводит к ошибке.

Наиболее типичный пример хука — уже знакомая нам функция `useState()`.

Все хуки, поддерживаемые React, будут рассмотрены в этой и последующих главах.

4.1.2. Пропы компонента

Проп[1] — это значение, передаваемое текущему компоненту его родителем. Может быть любым и принадлежать любому типу (в том числе функциональному или объектному).

Компонент может принимать произвольное количество пропов — в том числе ни одного (как, например, компонент приложения).

Каждый проп, получаемый текущим компонентом, должен иметь уникальное в пределах компонента имя, по которому будет выполняться обращение к этому пропу.

Все пропы, полученные компонентом, объединяются в простой объект JavaScript, который передается функции, реализующей этот компонент, с единственным параметром. Этому параметру обычно дают имя `props`.

Чтобы в коде компонента получить значение, переданные через проп с заданным именем, достаточно обратиться к одноименному свойству простого объекта, полученного через этот параметр.

В качестве примера рассмотрим компонент `TodoItem2`, принимающий три пропа: `todo`, `setDone` и `del` (листинг 4.2).

---

[1] Калька с английского prop — сокращения от property, свойство.

Листинг 4.2. Усовершенствованный компонент дела `TodoItem2`

```
export default function TodoItem2(props) {
    return (
        <div className="box">
            <h2>{props.todo.title}</h2>
            <p>{props.todo.desc}</p>
            <p>
                <button className="button is-success"
                        onClick={e => {props.setDone(item.key)}}
                >
                    Выполнено
                </button>
                <button className="button is-danger"
                        onClick={e => {props.del(item.key)}}
                >
                    Удалить
                </button>
            </p>
        </div>
    );
}
```

Если принимаемых компонентом пропов немного, для удобства их извлечения можно применить деструктурирование объектов:

```
export default function TodoItem2({ todo, setDone, del }) {
    return (
        <div className="box">
            <h2>{todo.title}</h2>
            <p>{todo.desc}</p>
            . . .
        </div>
    );
}
```

Это, кстати, позволяет указать у каких-либо пропов значения по умолчанию, присваиваемые им, если в родителе у этих пропов значения не были заданы. Пример указания у пропов `setDone` и `del` в качестве значений по умолчанию «пустых» функций:

```
export default function TodoItem2({ todo, setDone=()=>{}, del=()=>{} }) {
    . . .
}
```

Пропы — это одна из разновидностей *реактивных данных* (данных, изменение которых автоматически обновляет компонент).

Передача значений компоненту через пропы называется *спуском данных*.

Внимание!

Изменять значения пропов в коде компонента настоятельно не рекомендуется, т. к. это теоретически может нарушить работу компонента.

4.1.3. Рендеринг компонента

Рендерингом называется вывод компонента на страницу. Рендеринг выполняет сам React, основываясь на содержимом компонента, которое описывается объектной структурой и возвращается функцией, реализующей компонент.

Рендеринг бывает:

- *первоначальный* — выполняется при монтировании (первом отображении) компонента на странице.

  При первоначальном рендеринге React, помимо вывода содержимого компонента на странице, также создает в памяти его компактное представление — *виртуальную DOM*. Она потребуется при последующих обновлениях компонента, чтобы выяснить, какие элементы, входящие в состав содержимого компонента, требуется изменить, а какие — нет;

- *повторный* — выполняется при обновлении (повторном отображении) компонента.

  При повторном рендеринге React сравнивает новое содержимое компонента с текущим, которое описано виртуальной DOM, сохраненной в памяти ранее, при первоначальном рендеринге. При этом обновляются только изменившиеся элементы содержимого, за счет чего обновление компонента производится максимально быстро.

Рендеринг компонента в обоих случаях выполняется после завершения выполнения функции, представляющей этот компонент.

4.1.3.1. Объектное описание содержимого компонента. Фрагменты

Содержимое компонента описывается в функции, реализующей этот компонент, в виде особого объекта на языке JSX и возвращается этой функцией в виде результата.

Объектное описание содержимого может представлять собой:

- произвольный тег (например, блок — тег `<div>`), вмещающий в себя все содержимое компонента и называемый *корневым*:

```
return (
    <div className="box">
        <h2>{props.todo.title}</h2>
        <p>{props.todo.desc}</p>
    </div>
);
```

  Вариант с корневым тегом — самый распространенный;

- фрагмент, вмещающий все содержимое компонента.

  Фрагментом называется особый встроенный в React компонент, который не выводится на странице и служит лишь вместилищем для какого-либо содержимого. Фрагмент реализован в виде функции `Fragment` из модуля `react`. Пример:

```
import { Fragment } from 'react';
. . .
return (
    <Fragment>
        <h2>{props.todo.title}</h2>
        <p>{props.todo.desc}</p>
    </Fragment>
);
```

  Поддерживается компактная запись фрагмента (при ее использовании явно импортировать функцию `Fragment` необязательно):

```
return (
    <>
        <h2>{props.todo.title}</h2>
        <p>{props.todo.desc}</p>
    </>
);
```

  Фрагменты используются в случаях, когда добавление в содержимое компонента лишних тегов может исказить разметку страницы.

  Фрагменты также могут задействоваться, чтобы объединить часть тегов с какой-либо целью (например, чтобы указать у них ключ — какое-либо уникальное значение, однозначно идентифицирующее элемент, подробности — в *главе 5*):

```
<dl>
    {props.list.map(item => (
        <Fragment key={item.key}>
            <dt>{item.title}</dt>
            <dd>{item.desc}</dd>
        </Fragment>
    ))}
</dl>
```

  У фрагментов, записанных в компактном формате, указать ключ нельзя;

- массив тегов.

  При этом не следует забывать, во-первых, необходимость разделения отдельных тегов в массиве запятыми, и, во-вторых, указывать у тегов произвольные ключи (поскольку фактически выполняется вывод последовательности, описанный в *разд. 5.1.4*).

  Пример:

```
return [
    <h2 key="a">{props.todo.title}</h2>,
```

```
        <p key="b">{props.todo.createdAt}</p>,
        <p key="c">{props.todo.desc}</p>,
        <p key="d"><img src={props.todo.image} /></p>
    ];
```

Возврат массива тегов также позволяет отказаться от корневого тега, который может нарушить разметку страницы.

4.1.3.2. Разделение кода, выполняющего рендеринг

В функции компонента можно объявить вложенные функции, возвращающие в качестве результата части содержимого компонента, и вызывать эти функции в нужных местах кода, создающего содержимое. Это позволит сделать код более наглядным и упростить его сопровождение.

В листинге 4.3 показан код компонента перечня дел `TodoDetail2`, в котором использован принцип разделения кода, производящего рендеринг.

Листинг 4.3. Компонент перечня дел `TodoDetail2` с разделением кода, выполняющего рендеринг

```
import { useLoaderData } from 'react-router-dom';

export default function TodoDetail() {
    const todo = useLoaderData();

    const getDoneMark = () => {
        if (todo.done) // ..........................................  1
            return (
                <p className="has-text-success">
                    Выполнено
                </p>
            );
    }

    const getDesc = () => {
        if (todo.desc) // ..........................................  2
            return <p>{todo.desc}</p>;
    }

    const getImage = () => {
        if (todo.image) // .........................................  3
            return <p><img src={todo.image} /></p>;
    }

    return (
        <section>
            {getDoneMark()}
            <h1>{todo.title}</h1>
```

```
            <p>{todo.createdAt}</p>
            {getDesc()}
            {getImage()}
        </section>
    );
}
```

Такой код выглядит нагляднее, в том числе и за счет того, что для формирования содержимого компонента применялись более привычные JavaScript-программистам языковые конструкции, в частности условные выражения (поз. 1–3 в листинге 4.3).

4.1.3.3. Предотвращение рендеринга

Если какой-либо компонент должен выводить содержимое только в случае выполнения заданного условия, предотвратить его вывод, если условие не выполнено, можно, вернув из функции, реализующей компонент, какое-либо «пустое» значение: пустую строку, `null` или `undefined`. Пример:

```
export default function Message({ message }) {
    . . .
    if (message)
        // Если текст сообщения указан, выводим компонент
        return <p>{message}</p>;
    else
        // В противном случае не выводим ничего
        return null;
}
```

Также можно вообще ничего не возвращать из компонента — в этом случае будет неявно возвращено значение `undefined`:

```
export default function Message({ message }) {
    . . .
    if (message)
        return <p>{message}</p>;
}
```

4.1.3.4. Рендеринг произвольного содержимого

Можно написать компонент, который выводит произвольное содержимое, заданное в виде JSX-кода в теге этого компонента.

4.1.3.4.1. Рендеринг содержимого в одном месте компонента-родителя

Проще всего вывести заданное содержимое в каком-либо одном месте компонента-родителя.

Рассмотрим пример вывода компонента-спойлера `Spoiler`, у которого заголовок задается в пропе `title`, а содержимое — непосредственно внутри тега компонента:

```
<Spoiler title="Что такое React?">
    <p>Это клиентский веб-фреймворк.</p>
    <p>Он написан на языке JavaScript.</p>
</Spoiler>
```

Содержимое, заданное в теге компонента, передается в компонент через проп `children`. Если заданное содержимое:

- включает более одного элемента (как в приведенном примере) — оно представляется в виде обычного массива JavaScript;
- включает лишь один элемент — имеет объектный тип;
- вообще не указано — имеет неопределенный тип (`undefined`).

Фрагмент кода компонента `Spoiler`, в котором выполняется вывод содержимого:

```
export default function Spoiler({ title, children }) {
    . . .
    return (
        <div className="spoiler" . . . >
            <h6>{title}</h6>
            {children}
        </div>
    );
}
```

Если объем выводимого содержимого невелик, его можно задать в пропе `children` — это немного уменьшит объем кода. Единственный выводимый потомок можно указать как есть, например:

```
<Spoiler title="Что такое React"
         children={<p>Это клиентский веб-фреймворк.</p>} />
```

Если требуется указать несколько потомков, их нужно оформить в виде массива, не забыв указать у его элементов ключи:

```
<Spoiler title="Что такое React"
         children={[<p key="1">Это клиентский веб-фреймворк.</p>,
                    <p key="2">Он написан на языке JavaScript.</p>]} />
```

4.1.3.4.2. Рендеринг содержимого в разных местах компонента-родителя. Слоты

Иногда бывает необходимо вывести несколько фрагментов содержимого — в разных местах компонента. Например, компонент разметки может выводить в разных местах шапку сайта, панель навигации, основное содержимое и поддон. Для таких целей применяются *слоты* — пропы, каждый из которых хранит один из фрагментов выводимого содержимого.

Листинг 4.4 показывает код компонента `Layout`, который создает разметку страницы и выводит несколько фрагментов содержимого, заданных как в слотах, так и непосредственно в теге компонента.

Листинг 4.4. Компонент разметки `Layout`, выводящий разные фрагменты содержимого из разных слотов

```
export default function Layout(props) {
    return (
        <div>
            <header className="top">
                {props.heading}
            </header>
            <nav className="left">
                {props.navigation}
            </nav>
            <main className="right">
                {props.children}
            </main>
            <footer className="bottom">
                {props.footing}
            </footer>
        </div>
    );
}
```

Пример использования такого рода компонента:

```
function App() {
    return (
        <Layout heading={<h1>Todos</h1>}
                navigation={<NavigationPanel />}
                footing={<Footer developer="Тяп-Ляп Ltd" />}
        />
            <Greeting />
            <Advertising />
            <News />
        </Layout>
    );
}
```

4.1.3.4.3. Инструменты для работы с коллекцией потомков

React предоставляет класс `Children`, объявленный в модуле `react` и содержащий ряд статических методов для работы с выводимыми фрагментами содержимого, неважно, заданными в теге компонента или слотах. Вот эти методы:

- `count(<`*фрагмент содержимого*`>)` — **возвращает количество элементов в указанном** *фрагменте содержимого*:

```
import { Children } from 'react';
. . .
<div className="spoiler" . . .>
    . . .
```

```
    <div>Элементов в спойлере: {Children.count(props.children)}.</div>
</div>
```

- `map(<фрагмент содержимого>, <функция>)` — перебирает элементы из заданного *фрагмента содержимого* и для каждого его элемента вызывает указанную *функцию*, передавая ей два параметра: объект, представляющий очередной элемент содержимого, и индекс этого элемента.

  Объект, передаваемый первым параметром заданной *функции*, поддерживает следующие свойства:

  - `type` — имя тега или компонента, создающего элемент;
  - `props` — простой объект с пропами, переданными элементу (так, проп `className` содержит стилевые классы этого элемента, а проп `children` — его содержимое);
  - `key` — ключ или `null`, если таковой не был указан.

  Результаты, возвращенные *функцией*, помещаются в массив, который возвращается методом `map()` в качестве результата.

  Если *фрагмент содержимого* пуст, будет возвращено значение `undefined`.

  Пример компонента спойлера, выводящего только абзацы (теги `<p>`).

```
import { Children } from 'react';
. . .
<div className="spoiler" . . . >
    <h6>{title}</h6>
        {Children.map(
            children,
            item => item.type === 'p' ? item : ''
        )}
</div>
```

- `forEach()` — аналогичен методу `map()`, только не возвращает результата;
- `onlyChild(<фрагмент содержимого>)` — если указанный *фрагмент содержимого* содержит всего один элемент, возвращает этот элемент. Если же в составе *фрагмента* имеется больше одного элемента или этот *фрагмент* пуст, возбуждает исключение класса `Error`;
- `toArray(<фрагмент содержимого>)` — преобразует заданный *фрагмент содержимого* в обычный массив, содержащий элементы из этого *фрагмента*, и возвращает его.

> ***Внимание!***
>
> Начиная React 18, разработчики фреймворка предупреждают, что применение класса `Children` может снизить надежность кода[1]. Однако этот класс все еще поддерживается и не помечен как не рекомендованный к использованию.

[1] В документации по React буквально сказано: can lead to fragile code — может сделать код хрупким.

4.1.4. Состояния компонента

Состояние — это какое-либо значение, выработанное компонентом в процессе своего функционирования и принадлежащее непосредственно этому компоненту. Состояние может быть произвольным и относиться к любому поддерживаемому типу данных.

Компонент может создавать для работы сколько угодно состояний — в том числе и ни одного. Каждое состояние при создании получает уникальное имя.

Состояния компонента — это другая разновидность реактивных данных. При изменении любого состояния происходит автоматическое обновление компонента.

Для работы с состоянием применяется хук `useState()`, объявленный в модуле `react`. Он срабатывает в разные моменты работы компонента:

- при монтировании компонента — создает новое состояние, заносит в него указанное в вызове начальное значение, после чего выдает значение этого состояния и функцию для его изменения (подробнее о них будет рассказано позже);
- при обновлении — обращается к *уже существующему, созданному ранее* состоянию и выдает его значение вместе с функцией его изменения;
- перед размонтированием — удаляет состояние.

Формат вызова хука `useState()`:

```
useState([<начальное значение>|↳
         <функция, возвращающая начальное значение>])
```

Начальное значение может быть любого типа.

Функция, возвращающая начальное значение, не должна принимать параметров и должна возвращать значение любого типа. Следует отметить, что эта *функция* будет вызвана лишь один раз — при создании состояния (которое, как было сказано ранее, производится при монтировании компонента).

Если хук был вызван без параметров, создаваемое состояние получит начальное значение `undefined`.

Хук `useState()` возвращает массив, содержащий два элемента:

- текущее значение состояния, существовавшее на момент начала исполнения функции компонента.

  В процессе выполнения функции компонента это значение не изменяется;
- *сеттер состояния* — функция, которая при вызове изменяет это состояние. О ней мы поговорим позже.

Как правило, массив, возвращенный хуком, для удобства сразу деструктурируется, и его элементы присваиваются двум константам.

Пример создания состояния, хранящего признак того, нужно ли выводить меню навигации, с занесением в него начального значения `false`:

```
import { useState } from 'react';
. . .
const [showMenu, setShowMenu] = useState(false);
```

Пример создания состояния, содержащего массив планируемых дел, с использованием функции, возвращающей начальное значение;

```
const [todos, setTodos] = useState(
    () => needEmptyArray ?
        [] :
        ['Изучить React', 'Написать первое React-приложение']
);
```

Получить текущее значение состояния мы можем из первого элемента полученного массива, который при деструктурировании присвоили первой константе:

```
{showMenu && <div className="navigation-menu"> . . . </div>}
. . .
{todos.map(item => <TodoItem todo={item} />)}
```

Кстати, по соглашениям, имя константы, которой присваивается текущее значение состояния, становится именем самого состояния. Так, ранее мы создали и использовали состояния с именами `showMenu` и `todos`. А константе, хранящей сеттер состояния, дается имя, совпадающее с именем константы, в которой хранится текущее значение состояния, и предваренное префиксом `set`.

Чтобы изменить значение состояния, тем самым вызвав обновление компонента, нужно использовать сеттер состояния, извлеченный из второго элемента полученного массива. Вот формат его вызова:

```
<сеттер состояния>(<новое значение состояния>|↳
                   <функция, возвращающая новое значение состояния>)
```

Новое значение состояния должно отличаться от текущего значения — иначе состояние не будет изменено, а компонент не обновится.

Функция, возвращающая новое значение состояния, должна принимать единственным параметром текущее значение состояния и возвращать его новое значение.

Сам сеттер состояния не возвращает никакого результата.

Вот два примера изменения состояния `showMenu` на противоположное (в результате компонент обновится):

```
setShowMenu(!showMenu);
. . .
setShowMenu(currentState => !currentState);
```

При работе с состояниями следует иметь в виду два рассмотренных далее момента.

4.1.4.1. Состояние компонента: момент первый

Перед изменением состояния React сравнивает его текущее и новое значения и, собственно, изменяет состояние только в случае, если эти значения *различаются*. Так, следующий код не приведет к изменению состояния:

```
setShowMenu(showMenu);
```

Эта особенность React проявляется в полной мере, если в состоянии хранится значение какого-либо из ссылочных типов (например, массив или объект). Рассмотрим такой код:

```
todos.push('Реализовать маршрутизацию');
setTodos(todos);
```

В этом случае состояние не изменится, поскольку мы пытаемся занести в состояние тот же массив, который уже хранится в нем. Соответственно, компонент не обновится, и новое дело не появится в выведенном перечне.

Чтобы компонент обновился, необходимо создать копию массива, хранящегося в состоянии, и уже в эту копию добавить новое дело:

```
const newTodos = [...todos, 'Реализовать маршрутизацию'];
setTodos(newTodos);
```

То же самое касается изменения значения любого другого ссылочного типа, хранящегося в состоянии.

4.1.4.2. Состояние компонента: момент второй

Текущее значение состояния, выдаваемое хуком `useState()` в первом элементе возвращенного массива, *остается неизменным в течение всего времени выполнения функции компонента*. Это может привести к ряду проблем.

Предположим, мы создали состояние, хранящее число `0`, и решили трижды его инкрементировать (увеличить на единицу):

```
const [count, setCount] = useState(0);
. . .
useState(state + 1);
useState(state + 1);
useState(state + 1);
```

Мы предполагаем, что в результате в состоянии окажется число 3. Однако в действительности состояние будет содержать число 1. И вот почему...

Первый вызов сеттера `setCount()` запишет в состояние сумму текущего значения, равного `0`, и числа 1. Да, состояние при этом изменится, но константа `count` все еще будет хранить текущее значение состояния, то есть `0`. Поэтому второй вызов сеттера попытается занести в состояние также сумму `0` и 1 — то есть 1 (и состояние не изменится). Как и третий вызов сеттера...

Если требуется последовательно изменить значение состояния, следует вызвать его сеттер с указанием *функции, возвращающей новое состояние*:

```
useState(currentState => currentState + 1);
useState(currentState => currentState + 1);
useState(currentState => currentState + 1);
```

Вот теперь состояние `count` будет вполне ожидаемо хранить число 3.

Начиная с React 18, все операции изменения состояния, осуществляемые вызовами сеттеров, не выполняются сразу же, а ставятся в своего рода очередь и производят-

ся после выполнения функции компонента, перед его рендерингом. Это позволяет исключить излишние обновления компонента и тем самым повысить производительность приложения.

4.1.4.3. Как состояния связываются с компонентами?

Состояния, созданные в компоненте, хранятся отдельно от него. Для связывания каждого состояния с компонентом React использует особый указатель, состоящий из определения местоположения компонента в структуре DOM страницы, имени компонента и ключа (если он задан).

В большинстве случаев такой указатель однозначно связывает состояние с компонентом, в котором оно было создано. Однако в некоторых случаях возможны проблемы, наподобие рассматриваемой далее.

Предположим, что мы написали компонент `Counter`, содержащий кнопку и подсчитывающий количество щелчков на ней (листинг 4.5). Количество щелчков, равно как и номер счетчика (передаваемый через проп `num`), выводится непосредственно на кнопке.

Листинг 4.5. Компонент `Counter`, подсчитывающий количество щелчков на кнопке

```
import { useState } from 'react';

export default function Counter({ num }) {
    const [count, setCount] = useState(0);

    return (
        <p>
            <input type="button"
                   value={`Счетчик ${num}, щелчков: ${count}`}
                   onClick={() => setCount(count + 1)} />
        </p>
    );
}
```

В другом компоненте мы поместили подряд два экземпляра компонента `Counter` с разными номерами и набор из двух переключателей. Первый переключатель набора, изначально установленный, выводит первый экземпляр компонента `Counter`, второй переключатель отображает второй экземпляр. Пример:

```
const [num, setNum] = useState(0);
. . .
<p>
    <input type="radio" name="numcounter"
           check={num == 0}
           onChange={() => setNum(0)} />
    <label>Вывести первый счетчик</label>
    <br />
```

```
    <input type="radio" name="numcounter"
           check={num == 1}
           onChange={() => setNum(1)} />
    <label>Вывести второй счетчик</label>
</p>
{num == 0 ? <Counter num="1" /> : <Counter num="2" />}
```

Щелкнем несколько раз на кнопке, присутствующей в первом счетчике, изначально выведенном на экран. Посмотрим на количество щелчков, отображаемое непосредственно на кнопке. И установим второй переключатель, тем самым выведя на экран второй счетчик.

По идее, оба экземпляра компонента-счетчика должны работать независимо друг от друга. Следовательно, второй счетчик должен показать `0` щелчков. Однако он покажет столько же щелчков, сколько и первый (хотя номер счетчика, выводимый на кнопке, изменится).

Почему так происходит? Во-первых, оба наших счетчика являются экземплярами одного компонента — `Counter`. Во-вторых, они выводятся в одном и том же месте DOM страницы. Ранее было сказано, что указатель, который связывает состояние с компонентом, определяется местоположением компонента в DOM и его именем. Следовательно, если местоположения и имена обоих компонентов совпадают, созданное одним из них состояние будет связано с обоими компонентами.

Более того, с точки зрения React это не два экземпляра одного компонента последовательно выводятся в одном месте страницы, а *один-единственный компонент присутствует там постоянно*. Поэтому состояние при смене компонентов сохраняется.

Вот и выводят оба счетчика одно и то же число...

У этой проблемы есть два решения:

- вывести компоненты в разных местах DOM:

```
{num == 0 && <Counter num="1" />}
{num == 1 && <Counter num="2" />}
```

- указать у них разные ключи — поскольку указатели, связывающие состояния и компоненты, также содержат и ключи, если они заданы:

```
{num == 0 ? <Counter num="1" key="counter1" /> :
            <Counter num="2" key="counter1" />}
```

4.2. Дополнительные инструменты для написания компонентов

В этом разделе описаны программные инструменты, позволяющие реализовать специфическую функциональность компонентов.

4.2.1. Управление элементами веб-страниц. Рефы

В ряде случаев бывает необходимо получить доступ к элементу страницы, входящему в состав содержимого компонента, чтобы выполнить какие-либо действия над этим элементом. Для такого случая React предоставляет *рефы*[1] — своего рода ссылки на элементы страниц. Рефы бывают двух видов.

4.2.1.1. Объектные рефы

Объектный реф представляется в виде объекта и очень прост в использовании.

Чтобы работать в коде компонента с элементом страницы посредством объектного рефа, следует выполнить следующие действия:

1. Создать объектный реф — вызовом хука `useRef()` из модуля `react` согласно формату:

   `useRef(<начальное значение>)`

   Указанное *начальное значение* заносится в реф непосредственно при его создании. Если реф будет применяться для получения доступа к элементу страницы, можно задать любое начальное значение (например, `undefined`).

   Хук `useRef()` возвращает созданный реф, представленный особым объектом. Этот реф следует присвоить какой-либо переменной или константе.

   Объектный реф создается при монтировании компонента и удаляется при размонтировании. При обновлении компонента используется созданный ранее реф.

2. Присвоить созданный реф параметру `ref` нужного элемента страницы.
3. Извлечь ссылку на объект DOM, представляющий этот элемент страницы, из свойства `current` созданного ранее объекта рефа.

Получив таким образом элемент страницы, можно обращаться к его свойствам и методам.

В листинге 4.6 показан код компонента видеопроигрывателя `VideoPlayer`, использующего объектный реф для управления воспроизведением.

Листинг 4.6. Компонент видеопроигрывателя `VideoPlayer`, использующий объектный реф

```
import { useRef } from 'react';

export default function VideoPlayer() {
    const videoRef = useRef(undefined); // ..........................  1

    return (
        <div>
            <div>
```

[1] Калька с английского ref — сокращения от reference, ссылка.

```
                <video src="/movie.mp4" ref={videoRef} /> // .........  2
            </div>
            <div>
                <input type="button" value="Пуск"
                       onClick={() => videoRef.current.play()} /> // .  3
                <input type="button" value="Пауза"
                       onClick={() => videoRef.current.pause()} /> //   4
            </div>
        </div>
    );
}
```

Сначала создаем и сохраняем реф (поз. 1 в листинге 4.6). Далее связываем реф с элементом видеопроигрывателя — тегом `<video>` (поз. 2). Наконец, извлекаем из свойства `current` рефа ссылку на видеопроигрыватель и вызываем у него методы `play()` и `pause()` (поз. 3 и 4).

> ***Хранение произвольных значений в объектных рефах***
>
> Ранее говорилось, что объектный реф создается при монтировании компонента и существует все время, пока компонент присутствует на странице, вплоть до его размонтирования. Это дает возможность хранить в рефе произвольное значение все время существования компонента. Сохраненное в объектном рефе значение можно произвольно изменять в коде компонента, и изменение этого значения не приведет к его обновлению.
>
> Значение, которое следует сохранить в рефе, указывается в вызове хука `useRef()`:
>
> ```
> const valueRef = useRef(123);
> ```
>
> Далее в коде компонента доступ к хранящемуся в рефе значению можно получить через свойство `current` рефа:
>
> ```
> // Извлекаем из рефа текущее значение
> let val = valueRef.current;
> . . .
> // Заносим в реф другое значение. Компонент не обновляется.
> valueRef.current = 321;
> ```

4.2.1.2. Функциональные рефы

Функциональный реф (или *колбэк-реф*[1]) представляет собой вложенную функцию. Такие рефы несколько сложнее в использовании, чем объектные, однако позволяют реализовать дополнительную функциональность.

Чтобы создать функциональный реф, следует выполнить следующие действия:

1. Объявить в компоненте вложенную функцию, которая и реализует реф. Функция должна принимать единственный параметр и не должна возвращать результата.

   Эта функция за все время существования компонента будет вызвана дважды и получит с первым параметром:

[1] Калька с английского callback — обратный вызов.

- при монтировании компонента (первый вызов функции) — ссылку на объект DOM, представляющий элемент страницы, на который ссылается реф;
- при размонтировании компонента (второй вызов функции) — `null`.

Обычно эта функция сохраняет где-либо полученную ссылку на элемент страницы. Но она также может выполнять произвольные дополнительные действия.

Полученную ссылку не следует сохранять в локальной переменной, поскольку после завершения функции компонента эта переменная будет удалена. Ссылку нужно сохранить либо в глобальной переменной, либо, что предпочтительнее, в свойстве `current` заранее созданного объектного рефа.

2. Присвоить объявленную на шаге 1 функцию параметру `ref` нужного элемента страницы.

В листинге 4.7 показан код другой версии видеопроигрывателя, которая использует функциональный реф.

Листинг 4.7. Компонент видеопроигрывателя `VideoPlayer2`, использующий функциональный реф

```
import { useRef } from 'react';

export default function VideoPlayer2() {
    const videoRef = useRef(undefined) // ..........................  1
    const setVideoRef = el => {videoRef.current = el;} // ...........  2

    return (
        <div>
            <div>
                <video src="/movie.mp4" ref={setVideoRef} /> // ......  3
            </div>
            <div>
                <input type="button" value="Пуск"
                       onClick={() => videoRef.current.play()} /> // .  4
                <input type="button" value="Пауза"
                       onClick={() => videoRef.current.pause()} /> //   5
            </div>
        </div>
    );
}
```

Сначала создаем объектный реф, в котором будем хранить ссылку на видеопроигрыватель (поз. 1 в листинге 4.7). Далее объявляем функцию, которая занесет видеопроигрыватель в этот реф (поз. 2). Потом связываем эту функцию с тегом `<video>` (поз. 3). Наконец, извлекаем из созданного ранее рефа ссылку на проигрыватель и вызываем у него методы `play()` и `pause()` (поз. 4 и 5).

Ранее говорилось, что функция, реализующая функциональный реф, помимо сохранения ссылки на элемент в объектном рефе, может выполнять различные

дополнительные действия. Так, если у элемента страницы требуется отслеживать видимость, эта функция при монтировании компонента, получив ссылку на элемент, может регистрировать его в детекторе видимости, а при размонтировании, получив `null`, — убирать его регистрацию. Пример:

```
const observer = new IntersectionObserver( . . . );
. . .
const elRef = useRef(undefined);
const setElRef = el => {
    if (el) {
        observer.observe(el);
        elRef.current = el;
    } else {
        observer.unobserve(elReg.current);
        elRef.current = undefined;
    }
};
```

Начиная с React 19, из функции, реализующей реф, можно вернуть не принимающую параметров функцию очистки, которая будет вызвана при размонтировании компонента. В этом случае функция, реализующая функциональный реф, второй раз, при размонтировании, вызываться не будет. Пример:

```
const elRef = useRef(undefined);
const setElRef = el => {
    observer.observe(el);
    elRef.current = el;
    return () => {
        observer.unobserve(elReg.current);
        elRef.current = undefined;
    };
};
```

4.2.2. Порталы

Портал — это программный инструмент React, позволяющий вывести заданную часть содержимого компонента в произвольном элементе страницы приложения. Например, для вывода всплывающего сообщения можно воспользоваться порталом, созданным на основе другого элемента страницы, у которого средствами CSS задано абсолютное позиционирование.

Чтобы вывести нужную часть содержимого компонента в портале, в JSX-код компонента следует поместить вызов функции `createPortal()` из модуля `react-dom`, записанный в следующем формате:

```
createPortal(<часть содержимого, выводимая в портале>,
             <элемент страницы, на основе которого будет создан портал>[,
             <ключ портала>])
```

Часть содержимого, выводимая в портале, описывается на языке JSX и может представлять собой корневой тег, фрагмент или массив элементов (подробности — в *разд. 4.1.3.1*). *Элемент страницы, на основе которого будет создан портал*, задается в виде ссылки на соответствующий объект DOM. *Ключ портала* указывается лишь в том случае, если создаются несколько порталов.

В качестве результата функция `createPortal()` возвращает объект портала, который будет включен в объектное описание содержимого компонента.

Пример:

```
<!-- Страница приложения public\index.html -->
<!-- Блок, в котором выводится компонент приложения -->
<div id="root"></div>
. . .
<!-- Блок, на основе которого будет создан портал -->
<div id="portal"></div>

// Компонент TodoAdd
import { createPortal } from 'react-dom';
. . .
return (
    <section>
        <h1>Создание нового дела</h1>
        <form . . . >
            . . .
        </form>
    <section>
    {errorCode && createPortal(
        <p>Ошибка с кодом {errorCode}!</p>,
        document.getElementById('portal')
    )}
);
```

Можно создать несколько разных порталов и вывести в них разные части содержимого. Только в этом случае у каждого из создаваемых порталов следует указать *ключ*. Пример:

```
<!-- Страница приложения public\index.html -->
<div id="portal"></div>
<div id="portal2"></div>

// Компонент
return (
    . . .
    {createPortal(
        <p>Это портал 1</p>,
        document.getElementById('portal'), 'portal1'
    )}
```

```
    {createPortal(
        <p>Это портал 2</p>,
        document.getElementById('portal2'), 'portal2'
    )}
);
```

4.2.3. Принудительное обновление компонента

В *разд. 4.1.4* говорилось, что все изменения состояния, выполненные вызовами сеттеров, заносятся в особую очередь и производятся лишь после завершения выполнения функции компонента. Тогда же и производится обновление компонента, вызванное изменениями состояний.

Однако иногда возникает необходимость изменить состояния и сразу же после этого обновить компонент. Такое может понадобиться, например, при интеграции React с какими-либо другими библиотеками.

Принудительное обновления компонента после изменения состояний выполняется вызовом функции `flushSync()` из модуля `react-dom`:

```
flushSync(<функция, изменяющая состояния>)
```

функция, изменяющая состояния, не должна ни принимать параметров, ни возвращать результат. Обновление компонента будет произведено сразу после выполнения этой *функции*.

Результата функция `flushSync()` не возвращает.

Пример принудительного обновления компонента `App` сразу после изменения состояния `showMenu`:

```
import { flushSync } from 'react-dom';
. . .
flushSync(() => {
    setShowMenu(!showMenu);
});
```

> ***Внимание!***
>
> **Использование принудительного обновления компонента может сильно снизить производительность. Поэтому применять его следует только в случае крайней необходимости.**

4.2.4. Генерирование уникальных якорей

Якорь у элемента страницы указывается с помощью атрибута `id`, поддерживаемого всеми тегами. Все якоря в пределах страницы должны быть уникальными.

Хук `useId()` из модуля `react`, появившийся в React 18, возвращает гарантированно уникальный якорь, который можно указать у какого-либо элемента страницы. Этот якорь создается при монтировании компонента, остается неизменным при любых его обновлениях и уничтожается при размонтировании. Хук `useId()` вызывается без параметров.

Пример:

```
import { useId } from 'react';
. . .
const textId = useId();
. . .
<div className="control">
    <label htmlFor={textId}>{label}</label>
    <input id={textId}>
</div>
```

Полученный якорь можно использовать как основу для создания якорей для разных элементов страницы:

```
const cId = useId();
. . .
<form . . . >
    <div className="control">
        <label htmlFor={`${cId}-name`}>Имя пользователя:</label>
        <input id={`${cId}-name`}>
    </div>
    <div className="control">
        <label htmlFor={`${cId}-password`}>Пароль:</label>
        <input type="password" id={`${cId}-password`}>
    </div>
    . . .
</form>
```

4.3. Повышение производительности компонентов. Мемоизация

Мемоизация — это сохранение результата, возвращенного какой-либо функцией, для последующего использования в случаях, если значения переданных этой функции параметров не изменились с предыдущего вызова. В случае изменения параметров функция вызывается снова, и ее результат, опять же, сохраняется.

4.3.1. Мемоизация компонентов

Мемоизация компонентов (которые, как мы помним, также функции) позволяет предотвратить лишние обновления, поскольку мемоизированный компонент обновится только при изменении значений пропов, полученных им.

Мемоизированная функция компонента:

- при монтировании — вызывается в обязательном порядке, и возвращенный ею результат — содержимое компонента — сохраняется;
- при последующих обновлениях:
  - если ни один из пропов, полученных компонентом, не изменился — не вызывается, используется ранее сохраненный результат;

- если хотя бы один проп изменился — вызывается повторно, и возвращенный ею результат также сохраняется.

Однако если в мемоизированном компоненте изменится состояние, компонент обновится.

Для мемоизации компонента следует использовать функцию `memo()` из модуля `react`:

```
memo(<функция, реализующая компонент>[, <функция сравнения>])
```

Возвращаемым результатом станет новый компонент, заключающий в себе компонент, переданный первым параметром. Именно возвращенный компонент и следует использовать.

Пример:

```
import { memo } from 'react';

// Функция, реализующая компонент
function __TodoItem(props) {
    . . .
}

const TodoItem = memo(__TodoItem);
export default TodoItem;
```

Используются мемоизированные компоненты так же, как и обычные:

```
import TodoItem from './TodoItem.js';
. . .
{props.list.map((item) => (
    <TodoItem key={item.key} item={item} . . . />
))}
```

По умолчанию мемоизированные компоненты проверяют непосредственно значения полученных ими пропов. Если значением пропа является ссылка на массив или объект, компонент проверяет, не изменилась ли сама эта ссылка. Вследствие чего, если изменился не сам этот массив (объект), а лишь значение одного из его элементов (свойств), компонент посчитает, что соответствующий проп не изменился, и не обновится.

Для таких случаев в вызове функции `memo()` можно указать *функцию сравнения*, которая будет выяснять, изменились ли значения полученных им пропов. В качестве параметров *функция сравнения* должна принимать два простых объекта: первый — с предыдущими, второй — с текущими значениями пропов. Возвращать она должна `true`, если пропы *не изменились* (и соответственно компонент обновлять не нужно), и `false` — если они *изменились* (и компонент должен быть обновлен). Пример:

```
export default memo(TodoItem, (prevProps, nextProps) =>
    prevProps.item.done === nextProps.item.done;
);
```

4.3.2. Мемоизация результатов вычислений

Также можно мемоизировать любую функцию, выполняющую какие-либо сложные длительные вычисления и возвращающую их результат. Такая функция:

- при монтировании компонента — вызывается в обязательном порядке, и возвращенный ею результат сохраняется;
- при последующих обновлениях компонента:
  - если набор заданных значений, используемых этой функцией (*зависимостей*), не изменился — не вызывается, используется ранее сохраненный результат.

    В число зависимостей входят, прежде всего, параметры, получаемые этой функцией, и используемые в ее теле переменные;
  - если хотя бы одна зависимость из заданного набора изменилась — вызывается повторно, и возвращенный ею результат также сохраняется.

Мемоизация заданной *функции* на основании указанного *массива зависимостей* выполняется хуком `useMemo()` из модуля `react`:

```
useMemo(<мемоизируемая функция>, <массив зависимостей>)
```

Если передать пустой *массив зависимостей*, *функция* не будет мемоизироваться и станет выполняться при каждом обновлении компонента.

Пример:

```
import { useMemo } from 'react';

function doVeeeeeeeeeeeeryLongComputations(a, b, c) {
    . . .
}

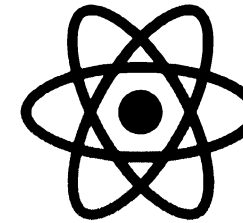
export default function SomeComponent({ x, y, z }) {
    . . .
    const result = useMemo(doVeeeeeeeeeeeeryLongComputations(x, y, z),
                           [x, y, z]);
    . . .
    return (
        . . .
        <p>{result}</p>
    );
}
```

Функция, задаваемая в вызове хука `useMemo()`, может возвращать результат любого типа, в том числе и фрагмент содержимого компонента, написанный на языке JSX. Это позволяет мемоизировать целые фрагменты компонента. Пример:

```
const result = useMemo(
    () => <p>{doVeeeeeeeeeeeeryLongComputations(x, y, z)}</p>,
```

```
    [x, y, z]
);
. . .
return (
    . . .
    {result}
    . . .
);
```

ГЛАВА 5



Язык JSX

Функция, реализующая компонент, должна возвращать его содержимое в виде структуры особых объектов, схожей с объектной моделью документа (DOM). Однако писать код, создающий такую структуру, очень трудоемко, к тому же наглядность этого кода стремится к нулю.

Поэтому разработчиками React для этой цели был создан специальный язык программирования JSX (JavaScript XML). Его синтаксис очень схож с синтаксисом языков HTML и XML.

Поскольку ни один веб-обозреватель не поддерживает JSX, написанный на нем код в процессе подготовки проекта к запуску транслируется программой Babel, входящей в состав проекта, в обычный JavaScript-код.

5.1. Основы JSX

Как говорилось ранее, синтаксис JSX очень похож на таковой у HTML. Основные принципы, по которым пишется JSX-код, приведены далее.

5.1.1. Синтаксис JSX

- Содержимое компонента описывается в виде набора вложенных друг в друга тегов, аналогичных тегам HTML.
- Для вывода тега в составе содержимого компонента используется следующий формат:

```
< <имя тега> [<параметр 1> <параметр 2> . . . <параметр n>] >
    <содержимое тега>
</ <имя тега> >
```

Имя тега должно быть набрано в нижнем регистре. Каждый из *параметров* тега записывается в формате:

```
<имя свойства>="<значение свойства>"
```

Для указания параметров используются *не атрибуты тегов*, поддерживаемые HTML, а *соответствующие им свойства классов DOM*. Например, для указания

стилевых классов у тега следует применять не атрибут тега `class`, а свойство `className`.

Пример:

```
<div className="navbar-brand">
    Todos
    <a href="/" className="navbar-burger">
        <span></span>
        <span></span>
        <span></span>
        <span></span>
    </a>
</div>
```

Вместо атрибута тега `for`, применяемого в теге `<label>`, следует использовать свойство `htmlFor`:

```
<label htmlFor="txtLogin">Имя пользователя</label>
<input id="txtLogin" />
```

Если у какого-либо тега надо указать свойство, хранящее значение `true`, можно записать лишь имя этого свойства (аналогично атрибуту HTML-тега без значения). Например, чтобы сделать поле ввода обязательным для заполнения, достаточно указать только свойство `required` без значения:

```
<input . . . required></input>
```

Если необходимо присвоить параметрам тега значения, хранящиеся в одноименных свойствах простого объекта, можно использовать деструктурирование:

```
const params = {className: 'table', id: 'todoList'};
. . .
<table {...params} /> . . . </table>
```

Пример передачи тегу параметров, полученных с пропами:

```
<TodoList list={todos} className="table" id="todoList" />
. . .
export default TodoList({ list, ...__others }) {
    . . .
    return (
        <table {...__others} /> . . . </table>
    );
}
```

- Все теги в JSX — парные, даже те, что в HTML являются одинарными, например `<img>`:

```
<img src="/logo.png" alt="Иллюстрация"></img>
```

Для записи одинарных, а также пустых парных тегов, можно использовать сокращенный формат:

```
< <имя тега> [<параметр 1> <параметр 2> . . . <параметр n>] />
```

Пример:

```
<img src="/logo.png" alt="Иллюстрация" />
```

> ***Внимание!***
> Присутствие в JSX-коде открывающего тега без закрывающей пары вызовет критическую ошибку при трансляции кода.

- Для вывода значения, получаемого в результате вычисления какого-либо *выражения* на языке JavaScript, применяется следующий формат:

```
{<вычисляемое выражение>}
```

Пример вывода в теге заголовка значения пропа `heading`:

```
<h1>{heading}</h1>
```

Пример вывода значения свойства `title` объекта, хранящегося в пропе `todo`:

```
<h2>{todo.title}</h2>
```

Выражение, вычисляющее выводимое значение, может быть более сложным:

```
<p>{todo.createdAt.toLocaleString()}</p>
```

- Чтобы присвоить значение, получаемое в результате вычисления *выражения* на языке JavaScript, какому-либо параметру тега, применяется формат:

```
<имя параметра>={<вычисляемое выражение>}
```

При этом *выражение* брать в двойные кавычки не следует. Если все же сделать это, *параметр* в качестве значения получит совершенно бесполезную строку с *выражением*.

Пример:

```
<a . . . className={showMenu ? 'navbar-burger is-active' : 'navbar-burger'}>
    . . .
</a>
```

- Чтобы привязать обработчик к событию какого-либо тега, следует присвоить свойству тега, соответствующему нужному событию, ссылку на функцию, которая и станет обработчиком. Пример привязки функции `showMenu()` к событию `click` в качестве обработчика:

```
<a . . . onClick={showMenu}>
    . . .
</a>
```

Более подробно обработка событий описана в *главе 6*.

- Для вывода в составе содержимого компонента другого компонента используется формат, аналогичный формату вывода тега:

```
< <тег компонента> [<проп 1> <проп 2> . . . <проп n>] >
    <содержимое компонента>
</ <тег компонента> >
```

Тег компонента совпадает с именем функции, реализующей компонент.

Внимание!

Имя компонента (и соответственно его тег) должно начинаться с большой буквы. Если набрать его с маленькой буквы, React посчитает, что выполняется вывод HTML-тега.

Каждый из *пропов* записывается в формате:

```
<имя пропа>="<значение пропа>"|{<вычисляемое выражение>}
```

Пример вывода компонента `Spoiler`:

```
<Spoiler title="Что такое React" opened={isSpoilerOpened}>
    <p>Это клиентский веб-фреймворк.</p>
    <p>Он написан на языке JavaScript.</p>
</Spoiler>
```

Для вывода пустых, не имеющих содержимого, компонентов можно применить сокращенный формат:

```
< <тег компонента> [<проп 1> <проп 2> . . . <проп n>] />
```

Пример:

```
<TodoAdd add={add} />
```

- JSX-код, занимающий две или более строки, должен быть заключен в круглые скобки:

```
return (
    <section>
        <h1>Создание нового дела</h1>
        <form>
            . . .
        </form>
    </section>
);
```

Если же JSX-код занимает лишь одну строку, заключать его в круглые скобки необязательно:

```
return <input type="button" value={caption} />;
```

5.1.1.1. Указание типа выводимого элемента во время выполнения

Тип выводимого элемента — имя тега или компонента-потомка — можно указать во время выполнения. Для этого следует:

1. Объявить какую-либо переменную с именем, начинающимся с большой буквы (это обязательно!).
2. Присвоить ей:
   - строку с именем тега — если требуется вывести HTML-тег;
   - ссылку на функцию компонента — если требуется вывести компонент.

3. Для непосредственно вывода — вставить в JSX-код конструкцию формата:

```
< <переменная> [<параметры или пропы>] >
    <содержимое>
</ <переменная> >
```

Также можно использовать сокращенный формат.

Пример вывода маркированного или нумерованного списка в зависимости от значения переменной `isNumbered`:

```
const Tag = isNumbered ? 'ol' : 'ul';
. . .
return (
    <Tag>
        <li>Первый пункт.</li>
        . . .
    </Tag>
);
```

Пример вывода компонента `TodoList` или `Empty` в зависимости от наличия элементов в массиве из состояния `todos`:

```
const Comp = (todos.length > 0) ? TodoList : Empty;
. . .
return <Comp list={todos} />;
```

5.1.2. Вывод значений

Все значения, выводимые на странице средствами JSX, отображаются как есть.

Исключение составляют лишь значения `true`, `false`, `null` и `undefined` — они вообще не выводятся на странице. Так, следующие шесть тегов не выведут ничего (точнее, пустой тег `<span>`, который никак не отобразится на странице):

```
<span></span>      <span>{true}</span>      <span>{false}</span>
<span />           <span>{null}</span>      <span>{undefined}</span>
```

Эту особенность React часто применяют для реализации условного вывода (будет описан чуть позже).

> ***Совет***
>
> Если попытаться вывести нотацию какого-либо HTML-тега, непосредственно записав его в JSX-код:
>
> ```
> Тег  выводит полужирный текст
> ```
>
> результатом будет критическая ошибка. Чтобы вывести тег, следует оформить его как строковое выражение JavaScript:
>
> ```
> Тег {''} выводит полужирный текст
> ```

Символы, недопустимые в содержимом HTML-тегов (например, двойные кавычки, знаки «меньше» и «больше»), присутствующие в выводимом значении, отображаются как есть:

```
<h1>Учим "React"</h1>                    // Результат: Учим "React"
<p>123 < 321</p>                         // Результат: 123 < 321
<p>{'123 < 321'}</p>                     // Результат: 123 < 321
```

> ***За кулисами***
>
> Это происходит потому, что перед выводом React преобразует недопустимые знаки в соответствующие им специальные символы HTML.

Недопустимые знаки также можно вывести, записав соответствующие им специальные символы:

```
<h1>Учим &quot;React&quot;</h1>          // Результат: Учим "React"
```

Если требуется создать в каком-либо элементе страницы сложное содержимое, описанное HTML-кодом, который хранится в какой-либо переменной, то выражение, подобное следующему:

```
const htmlContent = 'Это <strong>React</strong>!';
. . .
<div>{htmlContent}</div>
```

приведет к тому, что занесенный в элемент HTML-код будет просто выведен на странице. Попытка присвоить HTML-код свойству `innerHTML` элемента вообще ничего не даст — React заблокирует присваивание, и элемент окажется пустым.

Чтобы вывести в элементе содержимое, которое описывается HTML-кодом, следует создать простой объект со свойством `__html`, значением которого должен быть нужный HTML-код, и присвоить этот объект свойству `dangerouslySetInnerHTML` элемента:

```
const htmlContent = { __html: 'Это <strong>React</strong>!' };
. . .
<div dangerouslySetInnerHTML={htmlContent} />
```

5.1.3. Условный вывод

Условный вывод — это вывод в компоненте одного содержимого при выполнении заданного условия и вывод другого содержимого при его невыполнении. Частный случай условного вывода: вывод содержимого только при выполнении условия.

Обычные условные выражения JavaScript в JSX-код поместить нельзя. Вместо них для условного вывода следует применять один из программных механизмов, описанных далее.

- Тернарный оператор JavaScript. Пример привязки к гиперссылке разных стилевых классов в зависимости от значения состояния `showMenu`:

  ```
  <a . . . className={showMenu ? 'navbar-burger is-active' : 'navbar-burger'}>
      . . .
  </a>
  ```

  Если при выполнении или невыполнении условия вообще ничего не следует выводить, из соответствующей части оператора следует вернуть любое из значений, не выводящихся на странице: `true`, `false`, `null` и `undefined`.

Пример вывода абзаца с текстом **Выполнено** только в том случае, если свойство `done` объекта из состояния `todo` хранит `true`:

```
{todo.done ? <p>Выполнено</p> : undefined}
```

- Особенности выполнения оператора логического И. Если его первый операнд имеет значение `true`, на странице будет выведено значение второго операнда. Если же первый операнд оператора имеет значение `false`, на странице вообще ничего выведено не будет. Пример:

```
{todo.done && <p>Выполнено</p>}
```

Следует помнить, что значение первого операнда должно быть строго логического типа. Если же тип значения первого операнда отличается от логического, на страницу будет выведено значение первого операнда. Например, при выполнении приведенного далее кода на странице появится значение свойства `length` массива дел, а не компонент `TodoList`, как предполагалось:

```
{todos.length && <TodoList list={todos} />}
```

А при выполнении следующего кода все будет работать как надо:

```
{(todos.length > 0) && <TodoList list={todos} />}
```

5.1.4. Вывод последовательностей

Для вывода элементов последовательностей (например, массивов или коллекций) использовать циклы нельзя — при попытке поместить цикл в JSX-код возникнет критическая ошибка. Однако можно использовать метод `map()`, поддерживаемый большинством последовательностей.

Пример вывода массива дел, хранящегося в состоянии `todos`:

```
<ul>
    {todos.map(item => <li key={item.key}>{item.title}</li>)}
</ul>
```

5.1.4.1. Ключи

Ключ — это уникальный идентификатор, назначаемый тегу или вложенному компоненту, в котором выводится элемент какой-либо последовательности, и связывающий этот тег (вложенный компонент) с выводимым в нем элементом.

Ключ задается с помощью параметра `key`, который указывается непосредственно в теге (вложенном компоненте).

В качестве ключей следует использовать какие-либо гарантированно уникальные значения, хранящиеся в составе элементов выводимой последовательности: названия сущностей, временны́е отметки их создания, числовые или строковые идентификаторы (так, строковые идентификаторы используются в качестве ключей в приложении списка дел, разработанном в *главах 1* и *2*). В крайнем случае в качестве ключей можно использовать индексы элементов — но только если порядок элементов в последовательности не меняется.

Пример задания ключей у пунктов списка, выводящего перечень дел, был приведен в *разд. 5.1.4*.

Пример указания ключей у вложенных компонентов:

```
{todos.map(item => <TodoItem key={item.key} item={item} />)}
```

React использует ключи, чтобы определить, были ли элементы последовательности удалены, перемещены или добавлены, и при обновлении компонента соответственно удаляет, перемещает или создает соответствующие им теги или вложенные компоненты. Если ключи не указать, фреймворк будет вынужден удалить и повторно создать все содержимое обновляемого компонента, что негативно скажется на производительности. Кроме того, отсутствие ключей у тегов (вложенных компонентов), выводящих элементы последовательности, приведет к выводу сообщения о некритической ошибке.

> ***За кулисами***
>
> React при обновлении компонента старается по возможности внести в сформированную ранее DOM как можно меньше изменений. Так, если при обновлении компонента выяснится, что изменилось лишь содержимое одного тега, фреймворк изменит только содержимое этого тега, больше ничего не трогая.
>
> Чтобы реализовать оптимальное обновление, React активно использует виртуальную DOM, созданную при монтировании компонента.

5.2. Использование в компонентах внедренных элементов

В компонентах можно использовать любые внедренные элементы страницы — те, содержимое которых хранится в отдельных файлах: графические изображения, аудио- и видеоролики. Для этого следует:

- поместить файл, хранящий содержимое внедренного элемента, в папку src;
- выполнить импорт по умолчанию этого файла в модуле с кодом приложения, дав файлу произвольное имя;
- указать в теге, создающем внедренный элемент, в качестве ссылки на файл имя, под которым этот файл был импортирован ранее.

Пример вставки в содержимое компонента графического изображения из файла src\logo.png:

```
import logo from './logo.png';
. . .
<div>
    <img src={logo} alt="Логотип" />
</div>
```

Любой SVG-файл также можно импортировать как компонент React. Для этого следует выполнить множественный импорт такого файла, задав у него имя `ReactComponent` (это важно!) и произвольный псевдоним, обязательно с большой

буквы. После чего импортированный файл можно вставить в JSX-код как обычный React-компонент. Пример:

```
import { ReactComponent as Logo } from './logo.png';
. . .
<div>
    <Logo />
</div>
```

> ***За кулисами***
>
> В процессе подготовки приложения к запуску сборщик Webpack:
>
> - загрузит в память все файлы с содержимым внедренных элементов, которые были импортированы в JavaScript-модулях (вместе с файлами, хранящими связку);
> - переименует эти файлы, добавив к их именам хеши содержимого (чтобы сделать их имена уникальными для корректного сохранения в кеше веб-обозревателя).
>
> Если размер какого-либо файла меньше значения переменной окружения `IMAGE_INLINE_SIZE_LIMIT`, этот файл будет преобразован в data URL и вставлен непосредственно в JavaScript-код.

Если проект включает множество файлов с внедренными элементами, процесс сборки может отнять много времени. Ускорить его можно, исключив файлы с внедренными элементами из процесса сборки. Для этого нужно:

- переместить такие файлы (все или частично) из папки src в папку public;
- в качестве ссылок на эти файлы использовать их интернет-адреса, заданные относительно папки public.

  В начале каждого такого интернет-адреса следует указать адрес корневой папки приложения, хранящийся в переменной окружения `PUBLIC_URL`, например:

  ```
  <img src={process.env.PUBLIC_URL + 'logo.png'} alt="Логотип" />
  ```

Так же следует поступить, если какой-либо из файлов, хранящих содержимое внедренного компонента, должен иметь строго определенное имя, которое ни в коем случае не должно меняться.

5.3. Оформление компонентов

У компонента можно указать оформление, отличное от оформления, заданного в таблице стилей приложения (см. *разд. 3.3.3*).

5.3.1. Таблицы стилей компонентов

Оформление компонента можно записать в *таблице стилей компонента*.

По соглашениям файл с таблицей стилей компонента должен иметь то же имя, что и файл с модулем компонента (так, файл с таблицей стилей компонента `App` будет назван src\App.css). Импорт таблицы стилей компонента выполняется в модуле компонента, например:

```
import './App.css';
```

Таблица стилей компонента должна включать стили, непосредственно действующие на содержимое компонента. В селекторах таких стилей следует использовать стилевые классы. Пример:

```
{* Таблица стилей src\App.css *}
.App {
  text-align: center;
  margin: 10pt;
}
.App .bordered {
  border: 1px solid black;
}
.App p .redtext {
  color: red;
}
```

Эти стилевые классы позже используются для привязки стилей к элементам, составляющим содержимое компонента:

```
<div className="App">
    <p className="bordered">Текст в рамке</p>
    <p><span className="redtext">Красный</span> текст</p>
</div>
```

Как и в случае таблицы стилей приложения, таблицы стилей компонента можно писать как на языке CSS, так и на языке SASS (подробности — в *разд. 3.3.3.2*).

5.3.1.1. CSS-модули

CSS-модуль — это таблица стилей, в которой все стилевые классы и имена анимаций имеют локальную область действия. Это значит, что разные CSS-модули могут содержать разные стили с одинаковыми селекторами.

CSS-модули должны:

- содержать лишь стили с селекторами в виде стилевых классов;
- храниться в файлах с именами, заканчивающимися суффиксом .module.

Чтобы использовать CSS-модуль, следует выполнить его импорт по умолчанию, указав произвольное имя (обычно — `styles`). Импортированный CSS-модуль будет представлен в памяти как простой объект со свойствами, имена которых совпадают со стилевыми классами, а значения являются соответствующими стилями. Чтобы использовать один из стилей, содержащихся в импортированном CSS-модуле, достаточно указать в свойстве `className` нужного элемента обращение к одноименному свойству упомянутого ранее простого объекта.

Пример:

```
{* CSS-модуль src\TodoList.module.css *}
.heading {
  font-size: 48pt;
}
```

```
.list {
  width: 100%;
  border: thin solid grey;
}

// Компонент TodoList
import styles from './TodoList.module.css';
. . .
<h1 className={styles.heading}> . . . </h1>
<table className={styles.list}> . . . </table>
```

> ***За кулисами***
>
> В процессе подготовки приложения к запуску Webpack даст всем стилевым классам, присутствующим в каждом CSS-модуле, гарантированно уникальные наименования, чтобы стили из разных CSS-модулей не «смешивались» друг с другом.

5.3.2. Задание встроенных стилей CSS

Встроенные стили у элементов в JSX задаются посредством свойства `style`. Его значением должен быть простой объект, свойства которого соответствуют задаваемым атрибутам стиля и хранят значения, которые следует дать этим атрибутам стиля. Имена этих свойств должны совпадать с именами соответствующих атрибутов стиля, набранными без дефисов, в которых каждое слово имени, кроме первого, должно быть набрано с большой буквы (стиль именования *camelCase*). Например: атрибуту стиля `color` должно соответствовать свойство `color`, атрибуту `font-size` — свойство `fontSize`, а атрибуту `border-top-style` — свойство `borderTopStyle`.

Пример указания встроенного стиля:

```
const style1 = {
    color: 'red',
    fontSize: '24pt',
    borderBottom: '5px solid black'
};
. . .
return <p style={style1}>Вы ввели неправильное значение!</p>;
```

Значения атрибутов стиля, требующих указания единицы измерения размера, можно задать в числовом виде — тогда React автоматически добавит им обозначение `px` (пикселы):

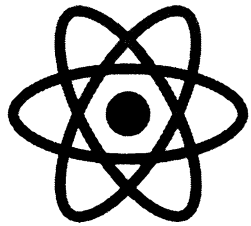
```
const style2 = { width: 80 };
```

Если у значения атрибута стиля нужно указать другую единицу измерения, ее обозначение следует добавить явно:

```
const style3 = { width: '80vh' };
```

К значениям атрибутов стиля, не требующих указания единицы измерения (например, `zIndex`), никакие обозначения добавляться не будут.

ГЛАВА 6



Обработка событий

Обработка событий в React выполняется практически по тем же принципам, что и в DOM. Сначала в теле компонента объявляется функция, которая станет обработчиком события. После чего ссылка на эту функцию присваивается соответствующему обрабатываемому событию параметру тега, в котором событие должно обрабатываться.

6.1. Написание обработчиков событий

Обработчики событий в React пишутся в виде:

- вложенных функций:

```
export default function Spoiler(props) {
    . . .
    function handleHeaderClick(evt) {
        . . .
    }
    . . .
    return (
        <div>
            <h6 onClick={handleHeaderClick}>Спойлер</h6>
            . . .
        </div>
    );
}
```

- стрелочных функций, присвоенных каким-либо локальным переменным:

```
export default function TodoAdd() {
    . . .
    const handleFormSubmit = evt => {
        . . .
    }
    . . .
```

```
    return (
        . . .
        <form onSubmit={handleFormSubmit}>
            . . .
        </form>
        . . .
    );
}
```

- стрелочных функций, записываемых непосредственно в тегах:

```
export default function TodoAdd() {
    . . .
    return (
                        . . .
                        <input . . .
                               onChange={e => setTitle(e.target.value)} />
                        . . .
    );
}
```

Однако при использовании этого подхода для каждого такого тега будет создан отдельный объект класса `Function`, представляющий стрелочную функцию. Если на странице создается много компонентов, хранение этих объектов функций может отнять много памяти.

В любом случае функции-обработчику в качестве параметра передается объект события.

Обработчикам событий любого типа по соглашению даются имена следующего формата:

`handle<`*наименование элемента страницы*`><`*имя события*`>`

Наименованием элемента страницы может быть как имя тега, создающего его, так и краткое описание этого элемента. *Имя события* записывается в стиле camelCase. Примеры: `handleFormSubmit` (обработчик события `submit` веб-формы), `handleImageChange` (обработчик события `change` поля выбора файла), `handleHeaderClick` (обработка события `click` какого-либо тега, выводящего заголовок).

6.2. Привязка обработчиков к событиям

Чтобы привязать обработчик к событию, надо выполнить следующие шаги:

1. Записать в тег, в котором нужно обрабатывать то или иное событие, параметр с именем такого формата:

   `on<`*имя события, записанное в стиле camelCase*`>`

   Например, если требуется привязать обработчик к событию `click`, следует использовать параметр `onClick`, а для обработки события `keydown` — параметр `onKeyDown`.

2. Присвоить этому параметру ссылку на функцию, которая станет обработчиком события.

Пример:

```
<form onSubmit={handleFormSubmit}> . . . </form>
```

Не забываем взять выражение, выдающее ссылку на функцию, в фигурные скобки. В противном случае при выполнении приведенного кода параметр `onSubmit` получит в качестве значения строку `'handleFormSubmit'`, которая никак не может обрабатывать события, и возникнет ошибка.

6.2.1. Передача параметров обработчикам событий

Если обработчику события при вызове требуется передать какие-либо параметры, надо выполнить следующие действия:

1. Использовать в качестве обработчика стрелочную функцию, привязанную непосредственно к тегу.
2. Вызвать в теле этой функции обработчик события, передав ему нужные параметры (в число которых может входить объект события).

Пример передачи обработчику события двух параметров: объекта события и идентификатора:

```
<p>{item.title}</p>
<p><a onClick={e => delete(e, item.key)}>Удалить</a></p>
```

6.3. События, поддерживаемые React. Класс события React

React позволяет обрабатывать практически все события, предоставляемые DOM и зафиксированные в веб-стандартах.

Объект, хранящий сведения о возникшем событии и передаваемый обработчику, создается на основе класса `SyntheticBaseEvent`, предоставляемого фреймворком. Это сделано для того, чтобы сгладить различия в особенностях обработки событий у разных веб-обозревателей. Класс `SyntheticBaseEvent` содержит те же свойства, что и стандартный класс `Event` и производные от него классы (информацию об этих свойствах ищите в документации по DOM).

Общие свойства класса, поддерживаемые событиями всех типов: `bubbles`, `cancelable`, `currentTarget`, `defaultPrevented`, `eventPhase`, `isTrusted`, `target`, `timeStamp` и `type`. Общие методы: `preventDefault()` и `stopPropagation()`.

Класс также поддерживает свойство `nativeEvent`, хранящее изначальный объект события DOM на тот случай, если он зачем-либо понадобится. Еще он поддерживает два следующих метода[1]:

[1] Эти методы пришли в React из популярной ранее библиотеки jQuery.

- `isDefaultPrevented()` — возвращает `true`, если обработка события по умолчанию ранее была прервана вызовом метода `preventDefault()`, и `false` — в противном случае;
- `isPropagationStopped()` — возвращает `true`, если дальнейшее прохождение события было прервано вызовом метода `stopPropagation()`, и `false` — в противном случае.

Далее приведен список категорий поддерживаемых событий, и для каждой категории указаны перечни самих событий, дополнительных свойств класса `SyntheticBaseEvent`, хранящих сведения о них, и дополнительных методов этого же класса:

- **мышь** — `onClick`, `onContextMenu`, `onDoubleClick`, `onDrag`, `onDragEnd`, `onDragEnter`, `onDragExit`, `onDragLeave`, `onDragOver`, `onDragStart`, `onDrop`, `onMouseDown`, `onMouseEnter`, `onMouseLeave`, `onMouseMove`, `onMouseOut`, `onMouseOver` и `onMouseUp`. **Свойства**: `altKey`, `button`, `buttons`, `clientX`, `clientY`, `ctrlKey`, `dataTransfer`, `metaKey`, `movementX`, `movementY`, `pageX`, `pageY`, `relatedTarget`, `screenX`, `screenY` и `shiftKey`. **Метод** — `getModifierState()`;
- **колесико мыши** — `onWheel`. **Свойства**: `deltaMode`, `deltaX`, `deltaY` и `deltaZ`;
- **касания** — `onTouchCancel`, `onTouchEnd`, `onTouchMove` и `onTouchStart`. **Свойства**: `altKey`, `changedTouches`, `ctrlKey`, `metaKey`, `shiftKey`, `targetTouches` и `touches`. **Метод** — `getModifierState()`;
- **клавиатура** — `onKeyDown`, `onKeyPress` и `onKeyUp`. **Свойства**: `altKey`, `charCode`, `code`, `ctrlKey`, `key`, `keyCode`, `locale`, `location`, `metaKey`, `repeat`, `shiftKey` и `which`. **Метод** — `getModifierState()`;
- **фокус ввода** — `onBlur` и `onFocus`. **Свойство** — `relatedTarget`;
- **веб-формы и элементы управления** — `onChange`, `onInput`, `onInvalid`, `onReset` и `onSubmit`;
- **выделение** — `onSelect`;
- **интерфейс** — `onScroll`. **Свойства**: `detail` и `view`;
- **мультимедийные элементы** — `onAbort`, `onCanPlay`, `onCanPlayThrough`, `onDurationChange`, `onEmptied`, `onEncrypted`, `onEnded`, `onError`, `onLoadedData`, `onLoadedMetadata`, `onLoadStart`, `onPause`, `onPlay`, `onPlaying`, `onProgress`, `onRateChange`, `onResize` **(начиная с React 18)**, `onSeeked`, `onSeeking`, `onStalled`, `onSuspend`, `onTimeUpdate`, `onVolumeChange` и `onWaiting`;
- **графические изображения и общие события** — `onError` и `onLoad`;
- **трансформации** — `onTransitionEnd`. **Свойства**: `elapsedTime`, `propertyName` и `pseudoElement`;
- **анимация** — `onAnimationStart`, `onAnimationEnd` и `onAnimationIteration`. **Свойства**: `animationName`, `elapsedTime` и `pseudoElement`;
- **буфер обмена** — `onCopy`, `onCut` и `onPaste`. **Свойство** — `clipboardData`;

- **композиция** — `onCompositionEnd`, `onCompositionStart` **и** `onCompositionUpdate`. **Свойство** — `data`;
- **координатные устройства**[1] — `onGotPointerCapture`, `onLostPointerCapture`, `onPointerCancel`, `onPointerDown`, `onPointerEnter`, `onPointerLeave`, `onPointerMove`, `onPointerOut`, `onPointerOver` **и** `onPointerUp`. **Свойства:** `height`, `isPrimary`, `pointerId`, `pointerType`, `pressure`, `tangentialPressure`, `tiltX`, `tiltY`, `twist` **и** `width`;
- **спойлер HTML** — `onToggle`;
- **диалог HTML** — `onCancel` **и** `onClose`.

6.4. Фазы событий в React

В React, как и в DOM, все события проходят фазы погружения и источника, а всплывающие события — еще и фазу всплытия. Это позволяет, например, обрабатывать возникающие в компонентах-потомках события в их родителе.

Пример, демонстрирующий всплытие событий:

```
function Panel(props) {
    const handleDivClick = evt => {
        console.log(props.title);
    }

    return (
        <div onClick={handleDivClick}>
            <h6>{props.title}</h6>
            {props.children}
        </div>
    );
}

export default function SomeComponent() {
    return (
        <Panel title="Panel 1">
            <Panel title="Panel 2">
                <Panel title="Panel 3" />
            </Panel>
        </Panel>
    );
}
```

При щелчке на компоненте `Panel` наибольшего уровня вложенности (с заголовком `"Panel 3"`) в консоли веб-обозревателя появятся строчки:

[1] Устройства, позволяющие указывать местоположение какой-либо точки в виртуальном пространстве: стилусы, графические планшеты, а также мыши.

```
Panel 3
Panel 2
Panel 1
```

Здесь видно, что событие щелчка мышью, возникшее в глубже вложенном компоненте, всплыло сначала в его родитель (с заголовком `"Panel 2"`), а потом — в родитель его родителя (`"Panel 1"`).

Любопытно, что события всплывают даже из компонентов, содержимое которых выводится в порталах (см. *разд. 4.2.2*). Пример:

```
import { createPortal } from 'react-dom';

function OuterPanel(props) {
    . . .
    return createPortal(
        <div onClick={handleDivClick}>
            . . .
        </div>,
        document.getElementById('portal')
    );
}

export default function SomeComponent() {
    return (
        <Panel title="Panel 1">
            <OuterPanel title="Panel 2">
                <Panel title="Panel 3" />
            </OuterPanel>
        </Panel>
    );
}
```

При щелчке на компоненте `Panel` наибольшего уровня вложенности (с заголовком `"Panel 3"`) в консоли веб-обозревателя появятся те же строчки, что и в примере, приведенном ранее. Это продемонстрирует, что событие щелчка мышью успешно всплывает сквозь компонент `OuterPanel`, отображаемый в портале.

6.4.1. Обработка событий на фазе погружения

Существует возможность обрабатывать события не на фазе всплытия, как обычно, а на фазе погружения. Для этого функцию-обработчик события надо присвоить параметру с именем следующего формата:

on<*имя события, записанное в стиле camelCase*>Capture

Пример:

```
function Panel(props) {
    . . .
```

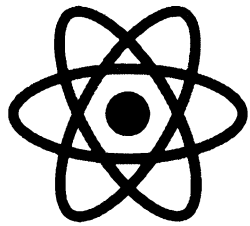
```
    return (
        <div onClickCapture={handleDivClick}>
            . . .
        </div>
    );
}
```

В этом случае при щелчке на компоненте наибольшего уровня вложенности (с заголовком `"Panel 3"`) в консоли веб-обозревателя появятся следующие строчки:

```
Panel 1
Panel 2
Panel 3
```

Сразу видно, что событие сначала возникает в наиболее «внешнем» компоненте (с заголовком `"Panel 1"`), далее погружается в его потомок (`"Panel 2"`) и, наконец, в потомок его потомка — компонент-источник (`"Panel 3"`).

ГЛАВА 7



Взаимодействие между компонентами

Разработчики React рекомендуют разделять отдельные части функциональности веб-приложения на как можно более простые и мелкие компоненты. Например, в качестве самостоятельных компонентов можно оформить составные части перечня дел: отдельную позицию перечня, заголовок дела, кнопку, помечающую дело как выполненное, и кнопку удаления дела. В качестве самостоятельных компонентов также можно оформить поле ввода регистрационного имени, кнопку выбора файла, кнопку-гамбургер и др.

Такой подход имеет два преимущества: первое — мелкие компоненты проще использовать повторно в более крупных и специализированных компонентах, и второе — простые компоненты удобнее отлаживать.

Понятно, что между отдельными компонентами, составляющими приложение, следует наладить взаимодействие, которое заключается в передаче между компонентами, во-первых, данных, во-вторых, функциональности и, в-третьих, управления.

7.1. Передача данных

Передача данных может производиться в двух направлениях:

- *спуск* — от текущего компонента к его потомкам;
- *подъем* — от текущего компонента к его родителю.

7.1.1. Спуск данных

Для спуска данных могут использоваться пропы или контексты. Последние более полезны для сквозного спуска.

7.1.1.1. Спуск данных посредством пропов. Простой и сквозной спуск

Спуск данных потомкам выполняется посредством пропов. Вот пример спуска массива дел от компонента `App` компоненту `TodoList` через проп `list`:

```
// Модуль src\App.js
export default function App() {
    . . .
    return (
        . . .
        <TodoList list={todos} />
        . . .
    );
}

// Модуль src\TodoList.js
export default function TodoList({ list }) {
    return (
        <table>
            {list.map(item => (
                <tr key={item.key}>
                    . . .
                </tr>
            ))}
        </table>
    );
}
```

Спуск данных бывает:

- *простой* — от текущего компонента к его непосредственному потомку (как было показано в приведенном примере);
- *сквозной* — от текущего компонента к потомку его потомка или еще глубже.

Вот пример сквозного спуска функции, помечающей дело как выполненное, от компонента `App` компоненту `DoneButton` через компонент `TodoList` посредством пропа `setDone`:

```
// Модуль src\App.js
export default function App() {
    . . .
    return (
        . . .
        <TodoList list={todos} setDone={setDone} />
        . . .
    );
}

// Модуль src\TodoList.js
export default function TodoList({ list, setDone }) {
    return (
        <table>
            {list.map(item => (
                <tr key={item.key}>
                    . . .
```

```
                    <DoneButton item={item} setDone={setDone} />
                    . . .
                </tr>
            ))}
        </table>
    );
}

function DoneButton({ item, setDone }) {
    return (
        <input type="button" value="Пометить как выполненное"
               onClick={() => setDone(item.key)} />
    );
}
```

7.1.1.2. Спуск данных посредством контекстов

Если какую-либо величину требуется спускать сразу множеству компонентов, в том числе многократно вложенных друг в друга, писать JSX-код, выполняющий сквозной спуск, станет слишком трудоемко. В этом случае лучше воспользоваться контекстом.

Контекст — это механизм React, позволяющий спускать произвольную величину произвольного типа (*нагрузку*) любому потомку текущего компонента, любому потомку его потомков и т. д. Один контекст может передавать только одну нагрузку.

Чтобы спустить какую-либо величину через контекст, надо выполнить следующие действия:

1. Создать контекст, через который будет передаваться эта величина, — вызовом функции `createContext()` из модуля `react`:

   `createContext(`*<значение нагрузки по умолчанию>*`)`

   Заданное *значение по умолчанию* сразу же будет дано нагрузке создаваемого контекста. Если в момент создания контекста это значение вычислить по какой-либо причине невозможно, следует задать `undefined` (а само значение задать позже способом, описываемым далее).

   Функция `createContext()` возвращает объект созданного контекста, который следует присвоить какой-либо переменной. Эта переменная должна иметь имя, записанное с большой буквы.

2. Поместить все компоненты, которые должны получать нагрузку созданного контекста, в провайдер контекста.

   Провайдер контекста — компонент, передающий контекст вместе с его нагрузкой всем своим потомкам. Также позволяет при необходимости указать другое значение у нагрузки.

Начиная с React 19, провайдер контекста записывается в JSX-коде посредством парного тега с именем следующего формата:

```
<переменная с объектом контекста>
```

Значение нагрузки указывается в пропе `value` провайдера контекста.

> **НА ЗАМЕТКУ**
>
> В версиях фреймворка, предшествовавших React 19, провайдер контекста записывался тегом формата:
>
> ```
> <переменная с объектом контекста>.Provider
> ```
>
> Этот формат все еще поддерживается для совместимости со старым кодом.

Каждый провайдер какого-либо контекста перекрывает значение нагрузки *того же* контекста, указанное при его создании, в вызове функции `createContext()`, но *только для своих потомков*. Компоненты, не являющиеся потомками провайдера, будут использовать значение, заданное при создании контекста.

Провайдеры одного и того же контекста можно вкладывать друг в друга. При этом значение нагрузки, заданное во вложенном провайдере, перекроет значение, заданное в провайдере-родителе.

3. В компоненте, использующем контекст, — подписаться на нужный контекст.

   Подписку на контекст, представленный указанным *объектом*, выполняет хук `useContext()` из модуля `react`:

   ```
   useContext(<объект контекста>)
   ```

   В качестве результата хук возвращает нагрузку заданного *контекста*.

   Начиная с React 19, для подписки на контекст можно использовать функцию `use()`, полностью аналогичную хуку `useContext()`[1].

   React ищет значение нагрузки заданного *контекста* сначала в родителе текущего компонента — в присутствующем там провайдере. Если провайдер *контекста* там отсутствует, фреймворк ищет провайдер в родителе родителя текущего компонента и т. д. Если же ни один провайдер нужного контекста не отыщется, будет использовано значение нагрузки по умолчанию, заданное в вызове функции `createContext()`.

   Нагрузка контекста — это *третья разновидность реактивных данных* (данных, изменение которых приводит к обновлению компонента).

Обычно все контексты объявляют в отдельном модуле и потом импортируют в те модули, где их требуется использовать.

В листинге 7.1 приведен код модуля src\contexts.js, в котором объявлены три контекста: `TodosContext` — который будет содержать в качестве нагрузки массив дел, `SetDoneContext` — для хранения функции пометки дела как выполненного, и `DelContext` — для хранения функции удаления дела.

---

[1] Впрочем, по мнению автора, хук `useContext()` удобнее, поскольку имеет «говорящее» имя.

Листинг 7.1. Модуль контекстов src\contexts.js

```
import { createContext } from 'react';
export const TodosContext = createContext(undefined);
export const SetDoneContext = createContext(undefined);
export const DelContext = createContext(undefined);
```

Вот пример передачи списка дел, функций `setDone()` и `del()` из компонента `App` компоненту `TodoList` посредством ранее созданных контекстов:

```
import { TodosContext, SetDoneContext, DelContext } from './contexts.js';
export default function App() {
    . . .
    return (
        <div className="container">
            . . .
            <main className="content px-6 py-6">
                <TodosContext value={todos}>
                    <SetDoneContext value={setDone}>
                        <DelContext value={del}>
                            <TodoList />
                        </DelContext>
                    </SetDoneContext>
                </TodosContext>
            </main>
        </div>
    );
}
```

А вот пример получения списка дел, функций `setDone()` и `del()`, переданных из компонента `App` в контекстах, компонентом `TodoList`:

```
import { useContext } from 'react';
import { TodosContext, SetDoneContext, DelContext } from './contexts.js';

export default function TodoList() {
    const list = useContext(TodosContext);
    const setDone = useContext(SetDoneContext);
    const del = useContext(DelContext);

    return (
        <table>
            {list.map(item => (
                <tr key={item.key}>
                    . . .
                    <button onClick={() => {setDone(item.key)}}>
                        &#9745;
                    </button>
                    . . .
```

```
                    <button onClick={() => {del(item.key)}}>
                        &#9746;
                    </button>
                    . . .
                </tr>
            ))}
        </table>
    );
}
```

Ранее говорилось, что значение нагрузки контекста будет извлекаться из провайдера, наиболее близкого к компоненту, использующему этот контекст. Вот пример, иллюстрирующий это:

```
// Простой объект с тремя темами оформления приложения
const themes = {
    white: { color: 'black', backgroundColor: 'white' },
    black: { color: 'white', backgroundColor: 'black' },
    grey: { color: 'lightgrey', backgroundColor: 'darkgrey' }
};

// Контекст, предназначенный для передачи обозначения текущей темы
// потомкам компонента приложения
const ThemeContext = createContext('white');

export default function App() {
    . . .
    return (
        . . .
        // В потомках, расположенных здесь, будет использована тема
        // по умолчанию - white
        <ThemeContext value="black">
            // Здесь будет использована тема black
            <ThemeContext value="darkgrey">
                // Здесь будет использована тема darkgrey
            </ThemeContext>
        </ThemeContext>
        // Здесь опять будет использована тема white
        . . .
    );
}
```

Есть еще один способ получить данные из контекста — использовать *потребитель контекста*, особый компонент, извлекающий из контекста нагрузку. Он генерируется автоматически при создании контекста и доступен из свойства `Consumer` объекта, представляющего контекст.

Компонент потребителя контекста записывается в JSX-коде посредством парного тега с именем следующего формата:

```
<объект контекста>.Consumer
```

В качестве его содержимого указывается функция, получающая с единственным параметром значение нагрузки и возвращающая какое-либо содержимое, которое должно выводиться на страницу, записанное в виде JSX-кода.

Пример:

```
export default function TodoList() {
    return (
        <table>
            <TodosContext.Consumer>
                {list => (
                    <SetDoneContext.Consumer>
                        {setDone => (
                            <DelContext.Consumer>
                                {del => (
                                    list.map(item => (
                                        <tr key={item.key}>
                                            . . .
                                        </tr>
                                    ))
                                )}
                            </DelContext.Consumer>
                        )}
                    </SetDoneContext.Consumer>
                )}
            </TodosContext.Consumer>
        </table>
    );
}
```

На заметку

Потребители контекста — более старый программный инструмент, нежели хук `useContext()`. Они не очень удобны в применении, поскольку вынуждают писать довольно громоздкий JSX-код (см. приведенный пример), вследствие чего в настоящее время применяются редко. Однако их еще можно встретить в старом коде.

7.1.2. Подъем данных

Подъем данных в React напрямую выполнить невозможно. Однако можно поднять произвольное количество любых значений опосредованно, выполнив следующие действия:

1. Объявить в компоненте-родителе, который должен получать данные от потомков, вложенную функцию, получающую передаваемые значения в качестве параметров. Эта функция может заносить полученные значения в какие-либо локальные переменные, выполнять заданные манипуляции с текущим компонентом и при необходимости возвращать результат.
2. Спустить ссылку на объявленную ранее функцию в компоненты-потомки посредством пропа или контекста.

3. Вызвать в компоненте-потомке полученную ранее функцию, передав ей с параметрами значения, предназначенные к подъему.

Вот пример подъема нового дела, созданного в компоненте `TodoAdd`, в компонент `App` посредством объявленной в последнем вложенной функции `add()`:

```
// Модуль src\App.js
export default function App() {
    . . .
    const add = deed => { . . . };
    . . .
    return (
        . . .
        <TodoAdd add={add} />
        . . .
    );
}

// Модуль src\TodoAdd.js
export default function TodoAdd(props) {
    . . .
    const handleFormSubmit = evt => {
        . . .
        const newDeed = { . . . };
        props.add(newDeed);
        . . .
    }
    . . .
    return (
        <form onSubmit={handleFormSubmit}>
            . . .
        </form>
    );
}
```

7.1.2.1. Мемоизация функций, передаваемых потомкам

Казалось бы, подъем данных выполнить несложно: надо лишь спустить потомку функцию и в коде потомка вызвать ее с указанием поднимаемых значений.

Проблема в том, что при каждом обновлении компонента-родителя эта функция (точнее, представляющий ее объект класса `Function`) создается заново. В результате каждый раз компонент-потомок будет получать с пропом новую функцию, что вызовет его обновление, как правило, совершенно не нужное. Причем обновляться компонент-потомок станет, даже будучи мемоизированным (мемоизация компонентов описана в *разд. 4.3.1*).

Устранить лишние обновления можно, мемоизировав функции, передаваемые потомкам. Мемоизированная таким образом функция:

- при монтировании компонента — создается и сохраняется для последующего использования;
- при обновлении компонента:
  - если ни одна из заданных зависимостей не изменилась — используется ранее сохраненная функция;
  - если какая-либо зависимость изменилась — создается повторно и, опять же, сохраняется.

Для мемоизации функции используется хук `useCallback()` из модуля `react`:

```
useCallback(<мемоизируемая функция>[, <массив зависимостей>])
```

В *массиве зависимостей* обычно указывают пропы, состояния и локальные переменные, используемые в теле *мемоизируемой функции*.

Хук `useCallback()` возвращает вновь созданную мемоизированную функцию, которую и следует передавать компонентам-потомкам.

Вот пример компонента `App`, мемоизирующего перед передачей потомкам функции `add()`, `setDone()` и `del()`:

```
import { useCallback } from 'react';

export default function App() {
    . . .
    const add = useCallback(deed => { . . . });
    const setDone = useCallback(key => { . . . });
    const del = useCallback(key => { . . . });

    return (
        . . .
        <TodoList list={todos} setDone={setDone} del={del} />
        <TodoAdd add={add} />
        . . .
    );
}
```

7.2. Передача функциональности. Рендер-пропы

Правила хорошего тона React-программирования предписывают разрабатывать компоненты как можно более универсальными, пригодными для повторного использования, в том числе в других проектах. В принципе следовать этим правилам несложно...

...ровно до того момента, когда очередной универсальный компонент должен, во-первых, генерировать часть своего содержимого на основе каких-либо данных (обычно получаемых с пропами или хранящихся в составе состояния), а во-вторых,

выводить эту часть содержимого по-разному в разных приложениях. Например, компонент перечня дел в одном приложении должен выводить каждую позицию в один столбец (содержащий заголовок дела), а в другом приложении — в три (заголовок, кнопка пометки дела как выполненного и кнопка удаления дела). Безусловно, можно писать свой компонент перечня дел для каждого приложения. Но намного удобнее использовать рендер-пропы.

Рендер-проп — это проп, через который компоненту спускается часть функциональности, уникальной для текущего проекта. Эта функциональность реализуется в виде обычной функции, которая:

- как правило, принимает параметры;
- на основе значений этих параметров генерирует фрагмент содержимого компонента;
- возвращает сгенерированный фрагмент в качестве результата.

Компонент может поддерживать произвольное количество рендер-пропов. Если рендер-проп в компоненте всего один, ему по соглашениям обычно дают имя `render`.

В листинге 7.2 показан код компонента `TodoList`, выводящего перечень дел и использующего рендер-проп `render` для отображения сведений об очередном деле.

Листинг 7.2. Компонент `TodoList`, использующий рендер-проп

```
export default function TodoList(props) {
    return (
        <section>
            <h1>Дела</h1>
            <table>
                {props.list.map(item => props.render(item))}
            </table>
        </section>
    );
}
```

Вот пример использования этого компонента для вывода перечня дел в один столбец, показывающий только заголовки дел:

```
<TodoList list={todos}
          render={item => (
              <tr key={item.key}>
                  <td>
                      {item.title}
                  </td>
              </tr>
          )}
/>
```

А вот пример вывода перечня дел в три столбца: заголовок, кнопка пометки дела как выполненного и кнопка удаления:

```
<TodoList list={todos}
          render={item => (
              <tr key={item.key}>
                  <td>
                      <Link to={`/${item.key}`}>
                          {item.done && <del>{item.title}</del>}
                          {!item.done && item.title}
                      </Link>
                  </td>
                  <td>
                      <DoneButton item={item} setDone={setDone} />
                  </td>
                  <td>
                      <DeleteButton item={item} del={del} />
                  </td>
              </tr>
          )}
/>
```

7.3. Передача управления

React позволяет из компонента-родителя управлять произвольным элементом содержимого компонента-потомка. Можно сказать, что в этом случае потомок передает родителю полномочия на управление своим элементом. Передать управление можно двумя способами.

7.3.1. Перенаправление рефов

Проще всего создать в родителе реф и спустить его потомку, который свяжет его с нужным элементом своего содержимого. После чего родитель сможет с легкостью управлять элементом содержимого потомка, на который ссылается спущенный им реф. Такой прием называется *перенаправлением рефа*.

Начиная с React 19, передать реф потомку можно через проп `ref` без каких бы то ни было дополнительных действий.

В листинге 7.3 приведен код компонента видеопроигрывателя `VideoPlayer`, который принимает реф от родителя.

Листинг 7.3. Компонент видеопроигрывателя `VideoPlayer`, принимающий реф от родителя

```
const VideoPlayer = props => (
    <video src={props.src} ref={props.ref} />
);
export default VideoPlayer;
```

Пример использования этого компонента:

```
import { useRef } from 'react';
import VideoPlayer from './VideoPlayer.js';

export default function VideoShowcase() {
    const videoRef = useRef(undefined);

    return (
        <div>
            <VideoPlayer src="movie.mp4" ref={videoRef} />
            <div>
                <input type="button" value="Пуск"
                       onClick={() => {videoRef.current.play();}} />
                <input type="button" value="Пауза"
                       onClick={() => {videoRef.current.pause();}} />
            </div>
        </div>
    );
}
```

На заметку

В версиях фреймворка, предшествовавших React 19, передать реф компоненту-потомку в пропе было невозможно. Приходилось предварительно преобразовывать потомок с помощью функции `forwardRef()` из модуля `react`. Эта функция поддерживается до сих пор для совместимости со старым кодом, но будет объявлена устаревшей в одной из следующих версий React.

7.3.2. Использование промежуточных обработчиков

Перенаправление рефа имеет серьезный недостаток. Получив доступ к элементу содержимого компонента-потомка, компонент-родитель сможет сделать с ним все что угодно. Так, компонент-родитель `VideoShowcase` сможет вызвать у видеопроигрывателя из состава компонента `VideoPlayer` (см. *разд. 7.3.1*) метод `remove()`, тем самым удалив этот элемент (что нарушит работу компонента и, возможно, всего приложения).

Поэтому в ряде случаев желательно ограничить набор программных инструментов, предоставляемых родителю для управления элементом содержимого потомка. Это можно выполнить, применив промежуточный обработчик.

Промежуточный обработчик — это простой объект JavaScript, который создается в потомке и содержит набор методов, предназначенных для управления элементом содержимого этого потомка. Для доступа к элементу эти методы могут использовать другой реф, созданный в потомке.

Для создания промежуточного обработчика применяется хук `useImperativeHandle()` из модуля `react`:

```
useImperativeHandle(<реф, полученный от родителя>,
                    <функция, возвращающая промежуточный обработчик>[,
                    <массив зависимостей>])
```

Функция, возвращающая промежуточный обработчик, не должна принимать параметров.

Массив зависимостей аналогичен тому, что передается хукам `useMemo()` (см. *разд. 4.3.2*) и `useCallback()` (см. *разд. 7.1.2.1*). Обычно в нем указывают пропы, состояния и локальные переменные, используемые в теле *функции, возвращающей промежуточный обработчик*.

Результата хук `useImperativeHandle()` не возвращает.

Промежуточный обработчик создается при монтировании компонента и сохраняется для дальнейшего использования. При обновлении компонента, если ни одна из приведенных в *массиве* зависимостей, не изменилась, используется ранее сохраненный промежуточный обработчик. Если же хотя бы одна зависимость изменилась, обработчик будет заново создан и также сохранен. При размонтировании промежуточный обработчик удаляется.

В листинге 7.4 приведен код компонента видеопроигрывателя `VideoPlayer2`, использующего промежуточный обработчик.

Листинг 7.4. Компонент видеопроигрывателя `VideoPlayer2`, использующий промежуточный обработчик

```
import { useRef, useImperativeHandle } from 'react';

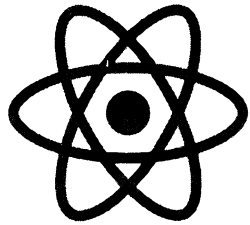
const VideoPlayer2 = props => {
    const __ref = useRef(undefined);

    useImperativeHandle(
        props.ref,
        () => {
            return {
                play() {
                    __ref.current.play();
                },
                pause() {
                    __ref.current.pause();
                },
            }
        }
    );

    return <video src={props.src} ref={__ref} />;
};
export default VideoPlayer2;
```

Теперь компоненту `VideoShowcase` доступны для вызова лишь методы `play()` и `pause()`, объявленные в промежуточном обработчике.

ГЛАВА 8



Работа с веб-формами и элементами управления

Никаких специфических средств для работы с элементами управления React не предоставляет. Все взаимодействие с ними производится исключительно стандартными инструментами DOM.

Однако в React 19 появился ряд весьма удобных инструментов для работы с веб-формами.

8.1. Работа с элементами управления

Сами разработчики React настоятельно советуют работать с элементами управления согласно следующим принципам:

- на каждый из элементов управления, в который будут заноситься данные, — создать отдельное состояние.

  В этом состоянии будет храниться копия значения, занесенного в элемент управления, — для упрощения его последующего извлечения, обработки и реализации какой-либо интерактивности (например, вывода или скрытия области редактирования при установке и сбросе флажка).

  При создании в это состояние нужно занести значение, которое должно выводиться в элементе управления изначально. У изначально пустых элементов управления надо указать пустую строку. Значения `undefined` и `null` в этом случае задавать не рекомендуется — во избежание нежелательных побочных эффектов;

- в элементе управления — вывести текущее значение соответствующего ему состояния.

  Для указания этого значения надо использовать следующие свойства:

  - у полей ввода, регуляторов и списков — `value`;
  - у областей редактирования — также `value` (а не запись значения в тег `<textarea>`, как в HTML);
  - у флажков и переключателей — `checked`;

- после ввода пользователем нового значения или изменения существующего — скопировать его из элемента управления в соответствующее состояние.

  Практически всегда это выполняется в обработчике события `change`, возникающем после изменения значения в элементе управления и потери им фокуса ввода.

  Перед занесением в состояние новое значение можно подвергать какой-либо обработке (например, приводить буквы к нижнему регистру или удалять начальные и конечные пробелы).

Пример работы с полем ввода адреса электронной почты в веб-форме входа:

```
import { useState } from 'react';
. . .
const [title, setTitle] = useState('');
. . .
<input value={title} onChange={e => {setTitle(e.target.value)}} />
```

Пример обработки адреса электронной почты перед занесением в состояние — приведение букв к нижнему регистру:

```
<input value={title}
       onChange={e => {setTitle(e.target.value.toLowerCase())}} />
```

Пример обработки флажка:

```
const [subscribe, setSubscribe] = useState(false);
. . .
<input type="checkbox" checked={subscribe}
       onChange={e => setSubscribe(e.target.checked)} />
```

Пример обработки набора переключателей можно увидеть в *разд. 4.1.4.1*.

Пример обработки списка:

```
const [gender, setGender] = useState('male');
. . .
<select value={gender} onChange={e => setGender(e.target.value)}>
    <option value="male">Муж.</option>
    <option value="female">Жен.</option>
</select>
```

При обработке поля выбора файлов в состоянии можно хранить:

- выбранный файл, уже преобразованный в представление, которое подходит для дальнейшей обработки (например, data URL — пример можно увидеть в листинге 1.4).

  Такой подход используется, если выбираемые в поле файлы относительно невелики по объему, могут быть считаны и преобразованы быстро;

- объект класса `File`, представляющий выбранный файл:

```
const handleImageChange = evt => {
    const cFiles = evt.target.files;
    setImage((cFiles.length > 0) ? cFiles[0] : undefined);
}
```

- объект класса `FileList`, представляющий коллекцию выбранных файлов:

```
const handleImageChange = evt => {
    setImage(evt.target.files);
}
```

Если для работы с данными, занесенными в веб-формы, не планируется использовать воздействия (см. *разд. 8.2.2*), все элементы управления рекомендуется обрабатывать описанным здесь способом.

8.2. Работа с веб-формами

Обрабатывать данные, занесенные в веб-формы, можно двумя способами. Первый способ предоставляет больше возможностей по работе с данными, но второй проще в реализации.

8.2.1. Обработка события *submit* веб-формы

Первый способ — работа с данными в обработчике события `submit` веб-формы. Это событие возникает непосредственно после нажатия кнопки отправки данных в веб-форме и перед собственно отправкой данных.

В этом обработчике сразу же следует отменить обработку события по умолчанию, вызвав у полученного объекта события метод `preventDefault()`. Если этого не сделать, веб-обозреватель отправит данные из веб-формы по текущему интернет-адресу, что вызовет перезагрузку приложения с потерей всех занесенных данных.

Пример:

```
import { useState } from 'react';

export default function TodoAdd() {
    const [title, setTitle] = useState('');
    const [desc, setDesc] = useState('');
    const [image, setImage] = useState('');
    . . .
    const handleFormSubmit = evt => {
        // Не забываем отменить обработку события по умолчанию
        evt.preventDefault();
        // Извлекаем данные, занесенные в веб-форму, из ранее созданных
        // состояний и сохраняем их (или обрабатываем иным образом)
        saveData({title, desc, image});
    };
    . . .
    return (
        <form onSubmit={handleFormSubmit}>
            . . .
        </form>
    );
}
```

Как говорилось ранее, этот способ позволяет реализовать практически любую логику обработки данных. Однако для его реализации придется писать дополнительный программный код.

8.2.2. Использование воздействий React

Воздействие React аналогично уже знакомому нам воздействию React Router (см. *разд. 1.9*), только обрабатывается не маршрутизатором, а самим фреймворком React. Поддержка таких воздействий появилась в React 19.

Воздействия React бывают двух видов: упрощенные и полнофункциональные.

8.2.2.1. Упрощенные воздействия React

Упрощенные воздействия React предоставляют ограниченные возможности в плане извлечения данных из веб-форм. Однако их проще писать.

Упрощенное воздействие представляет собой функцию, которая получает единственный параметр — объект класса `FormData`, хранящий занесенные в веб-форму данные. Результата она возвращать не должна. Допускается создание асинхронных упрощенных воздействий.

Для успешного выполнения упрощенного воздействия ссылку на него следует указать в параметре `action` веб-формы (тега `<form>`). Также допускается указывать ссылку на воздействие в параметре `formAction` кнопки отправки данных, которая может быть создана любым из тегов: `<input type="submit">`, `<input type="image">` или `<button>`.

Кроме того, у каждого элемента управления, в который заносятся данные, подлежащие обработке в воздействии, необходимо задать наименование (в атрибуте тега `name`).

Вот пример веб-формы входа, в которой для выполнения входа используется упрощенное воздействие и реализован вывод сообщений об ошибках:

```
import { useState } from 'react';
import { useNavigate } from 'react-router-dom';
import { login } from './api.js';

export default function Login() {
    const [error, setError] = useState(''); // ..........................  4
    const navigate = useNavigate(); // ..................................  7

    const actionHandler = async fd => { // ..............................  1
        const res = await login(fd); // .................................  2
        if (res)
            setError(res); // ........................................... 3
        else
            navigate('/'); // ........................................... 6
    };
```

```
    return (
        <section>
            <form action={actionHandler}> // ..............................  8
                <input type="email" name="email" /> // ....................  9
                . . .
                <input type="password" name="password" /> // ..............  10
                . . .
                <input type="submit" value="Войти" />
            </form>
            {error && <p>{error}</p>} // ..................................  5
        </section>
    );
}
```

Упрощенное воздействие `actionHandler` обрабатывает сведения о пользователе из веб-формы входа (поз. 1 в приведенном примере). Оно примет с единственным параметром объект класса `FormData`, содержащий данные из веб-формы.

В теле воздействия вызываем асинхронную функцию `login()` из модуля src\api.js, выполняющую вход, передавая ей данные из веб-формы (поз. 2). Функция вернет либо строковый код ошибки, либо, если вход был выполнен успешно, пустую строку.

Если функция `login()` вернула непустую строку, значит, возникла ошибка. Заносим возвращенный ею код ошибки в состояние `error` (поз. 3), которое создали ранее (поз. 4), чтобы вывести его на странице (поз. 5).

Если функция `login()` вернула пустую строку (то есть вход успешно выполнен), производим перенаправление на главную страницу. Для этого используем функцию (поз. 6), возвращенную хуком `useNavigate()` из модуля `react-router-dom` (поз. 7, этот хук описан в *главе 11*).

Ссылку на воздействие записываем в пропе `action` веб-формы (поз. 8). И не забываем указать наименования у обоих полей ввода (поз. 9 и 10).

Отметим, что мы больше не производим обработку элементов управления в стиле React, сохраняя заносимые в них значения в состояниях (см. *разд. 8.1*). В этом нет необходимости — фреймворк сам извлечет введенные данные, объединит их в объект и передаст воздействию. Однако если вводимые данные требуется подвергать валидации и (или) как-либо преобразовывать в процессе ввода, обработку элементов управления в стиле React все же придется выполнить.

Как мы только что убедились, применение воздействий заметно упрощает программирование. Однако если в веб-форме выбирается файл, и к содержимому этого файла требуется получить доступ на стороне клиента, придется использовать первый способ работы с веб-формами — поскольку извлечь содержимое выбранного пользователем файла из объекта класса `FormData` невозможно[1].

---

[1] Не совсем понятны мотивы разработчиков веб-стандартов, которые закрыли веб-сценариям доступ к выбранным пользователем файлам. Возможно, они хотели обезопасить пользователей от программ-шпионов...

Упрощенные воздействия имеет смысл применять в случае, если не требуется получать данные, присутствовавшие в веб-форме ранее, при предыдущей отправке данных. Если же эти данные нужны, следует использовать полнофункциональные воздействия (см. *разд. 8.2.2.2*).

8.2.2.1.1. Получение сведений о веб-форме

Сведения о веб-форме включают ее состояние (обрабатываются ли в настоящий момент данные из нее заданным воздействием или нет), введенные в нее данные и др.

Получить состояние веб-формы позволяет хук `useFormStatus()` из модуля `react-dom`. Он вызывается без параметров.

Хук возвращает простой объект со следующими свойствами:

- `pending` — `true`, если данные из веб-формы в настоящее время проходят обработку в воздействии, и `false` — в противном случае;
- `data` — объект класса `FormData`, содержащий данные из веб-формы;
- `method` — название HTTP-метода, указанного в пропе `method` тега веб-формы: `'get'` или `'post'`. По умолчанию: `'get'`;
- `action` — ссылка на функцию, реализующую воздействие, которая указана в пропе `action` веб-формы.

Объект, возвращенный хуком `useFormStatus()`, обновляется при изменении состояния веб-формы. Кроме того, он относится к реактивным данным — любое его изменение обновляет компонент.

Хук `useFormData()` следует вызывать в компоненте, являющимся дочерним по отношению к веб-форме.

В листинге 8.1 приведен код компонента кнопки отправки данных `SubmitButton`, который отслеживает состояние родительской веб-формы и соответственно меняет свою доступность.

Листинг 8.1. Компонент кнопки отправки данных `SubmitButton`, отслеживающий состояние родительской веб-формы

```
import { useFormStatus } from 'react-dom';
export default function SubmitButton(props) {
    const { pending } = useFormStatus();
    return <input type="submit" disabled={pending} {...props} />;
}
```

Этот компонент может быть использован следующим образом:

```
import SubmitButton from './SubmitButton.js';
. . .
<form action={actionHandler}>
    . . .
    <SubmitButton value="Войти" />
</form>
```

Если же вызвать хук `useFormStatus()` в том же компоненте, в котором находится веб-форма, чье состояние необходимо получить:

```
const { pending } = useFormStatus();
. . .
<form action={actionHandler}>
    . . .
    <input type="submit" disabled={pending} value="Войти" />
</form>
```

свойство `pending` простого объекта, возвращенного хуком, всегда будет хранить значение `false`.

8.2.2.2. Полнофункциональные воздействия React

Полнофункциональное воздействие React позволяет извлечь из веб-формы не только текущие данные, но и данные, присутствовавшие в ней на момент предыдущей отправки данных. В ряде случаев это может оказаться полезным. Однако писать такие воздействия несколько сложнее.

Сначала следует создать своего рода заготовку для полнофункционального воздействия — в виде функции, принимающей два параметра:

- предыдущее содержимое веб-формы — данные, которые присутствовали в ней на момент предыдущей отправки, или, если это первая отправка данных, — изначальное содержимое, переданное хуку `useActionState()` вторым параметром (подробности — чуть позже).

  Эти данные могут представляться значением любого типа: объектного (в том числе массивом), строкового, числового и др. Тип и формат этих данных задается значением, передаваемым вторым параметром хуку `useActionState()`;
- текущие данные, занесенные в веб-форму, — в виде объекта класса `FormData`.

Возвращать эта функция должна новое содержимое веб-формы в том же типе и формате, что и предыдущее содержимое, полученное ею с первым параметром. Возвращенное ею новое содержимое будет передано ей первым параметром при следующем вызове.

Готовое полнофункциональное воздействие на основе заданной *заготовки* создается вызовом хука `useActionState()` из модуля `react`:

```
useActionState(<заготовка для воздействия>, <изначальное содержимое веб-формы>)
```

Изначальное содержимое будет передано функции-*заготовке* первым параметром при первом вызове. Как говорилось ранее, оно может быть произвольного типа и формата.

Хук `useActionState()` возвращает массив из двух элементов:

- текущего содержимого веб-формы, возвращенного функцией-*заготовкой*.

  Текущее содержимое относится к реактивным данным — при его изменении компонент обновляется;

- функция, реализующая готовое воздействие. Эту функцию следует указать в пропе `action` тега веб-формы.

В листинге 8.2 показан код компонента счетчика `Counter`, аналогичного рассмотренному в *разд. 4.1.4.1* и использующему для подсчета щелчков полнофункциональное воздействие React.

Листинг 8.2. Компонент счетчика `Counter`, использующий полнофункциональное воздействие

```
import { useActionState } from 'react';

export default function Counter({ num }) {
    const [count, incrementCount] = useActionState(
        previousCount => previousCount + 1,
        0
    );

    return (
        <form action={incrementCount}>
            <input type="submit" value={`Щелчков: ${count}`} />
        </form>
    );
}
```

Поскольку в функции-заготовке создаваемого воздействия текущие данные из веб-формы не нужны, мы не указываем в этой функции второй параметр.

Пример веб-формы входа, аналогичный представленному в *разд. 8.2.2.1*, только использующий полнофункциональное воздействие React:

```
import { useActionState } from 'react';
import { useNavigate } from 'react-router-dom';
import { login } from './api.js';
import SubmitButton from './SubmitButton.js';

export default function Login() {
    const navigate = useNavigate();

    const [error, actionHandler] = useActionState(
        async (previous, fd) => {
            const res = await login(fd);
            if (res)
                return res;
            else
                navigate('/');
        },
        ''
    );
```

```
    return (
        <section>
            <form action={actionHandler}>
                . . .
            </form>
            {error && <p>{error}</p>}
        </section>
    );
}
```

В этом случае нам уже не нужен первый параметр функции-заготовки (однако его все равно придется указать).

При использовании полнофункциональных воздействий React также можно получать сведения о веб-форме инструментами, описанными в *разд. 8.2.2.1.1*.

Полнофункциональные воздействия React удобно применять, если по каким-либо причинам требуется доступ к предыдущему содержимому веб-формы. Если же предыдущие данные не нужны, лучше использовать упрощенные воздействия (см. *разд. 8.2.2.1*).

8.3. Валидация заносимых данных

Валидация — это проверка на корректность данных, заносимых в элементы управления, в первую очередь в поля ввода и в области редактирования.

Простейшую валидацию можно реализовать средствами HTML. Но, пользуясь средствами React, удается относительно просто реализовать программную валидацию.

Программная валидация данных в стиле React реализуется в несколько следующих шагов:

1. На каждый элемент управления, подвергаемый валидации, — создается отдельное состояние для хранения сообщения об ошибке ввода с пустой строкой в качестве изначального значения.

   Далее пустая строка будет обозначать отсутствие сообщения об ошибке (т. е. корректность введенных данных).

2. Возле каждого элемента управления (обычно под ним) отводится место под вывод сообщения об ошибке, содержащегося в соответствующем состоянии.

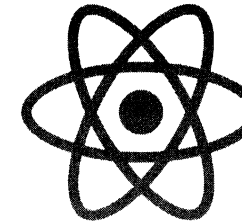
   Обычно сообщение выводится в виде абзаца (тега `<p>`) или блока (`<div>`). Если же сообщение об ошибке отсутствует, тег не выводится на экран.

   Может случиться так, что при выводе тега с сообщением об ошибке нарушится верстка страницы (и расположенные далее элементы страницы «уедут» вниз). В таком случае при отсутствии сообщения об ошибке следует вывести пустой тег, содержащий неразрывный пробел (специальный символ ` `), который никак не отобразится на странице.

3. Перед валидацией все заданные ранее сообщения об ошибках удаляются — занесением пустых строк в соответствующие состояния. Это нужно на тот случай, если пользователь допустил ошибку ввода ранее и теперь исправил ее.
4. Непосредственно перед обработкой данных — в обработчике события `submit` веб-формы или воздействии — производится собственно валидация.
   - Если очередное значение некорректно — в соответствующее состояние компонента заносится нужное сообщение об ошибке, и обработка данных прерывается. В результате компонент будет обновлен, и сообщение об ошибке, занесенное в состояние, появится на странице.
   - Если все значения корректны — данные пускаются в обработку.

Пример кода, выполняющего валидацию, можно увидеть в *разд. 2.6*.

ГЛАВА 9



Работа с внешними API

При разработке компонентов часто приходится обращаться к внешним по отношению к React программным интерфейсам. К таковым относятся, прежде всего, загрузчик данных (метод `fetch()` окна), многократные и однократные таймеры и прочие инструменты, предоставляемые классом окна `Window`.

9.1. Использование эффектов React

Код, обращающийся к внешним API, рекомендуется оформлять в виде эффектов. *Эффектом* называется функция, вызываемая строго после завершения выполнения функции, реализующей компонент, и перед его отображением на экране. Эффекты бывают трех видов.

9.1.1. Эффекты, вызываемые перед рендерингом компонента

Эффект, вызываемый перед рендерингом компонента, создается на основе заданной *функции* с применением хука `useEffect()` из модуля `react`:

`useEffect(<`*функция, реализующая эффект*`>[, <`*массив зависимостей*`>])`

Задаваемая *функция* не должна принимать параметров.

Задавать асинхронные *функции* не допускается. Если в коде эффекта необходимо выполнить асинхронный код, его следует оформить в виде вложенной функции, обычной или однократного вызова.

Результата хук `useEffect()` не возвращает.

Созданный эффект будет выполняться:

- при монтировании компонента — всегда;
- при очередном обновлении компонента — если *массив зависимостей*:
  - не указан или если задано значение `undefined` или `null` — всегда;

- указан и *не* пуст — только в случае изменения какой-либо из зависимостей, приведенных в *массиве*;
- указан и пуст — никогда.

Вот пример эффекта, загружающего массив дел с бэкенда:

```
import { useState, useEffect } from 'react';
import { getTodos } from './api.js';

export default function TodoList() {
    const [list, setList] = useState([]);

    useEffect(() => {
        (async () => {
            setList(await getTodos());
        })();
    });
    . . .
}
```

Загрузку массива дел выполняет асинхронная функция `getTodos()` из модуля src\api.js. Загруженный массив помещаем в специально созданное для этого состояние `list`.

Поскольку при создании эффекта мы в вызове хука `useEffect()` не указали массив зависимостей, созданный эффект будет вызываться при монтировании и при каждом обновлении компонента. Собственно, это то, что нам нужно.

Отметим, что асинхронную часть кода эффекта, в которой вызывается функция `getTodos()`, мы оформили в виде асинхронной функции однократного вызова, поскольку сама функция, реализующая эффект, должна быть синхронной.

Пример эффекта, загружающего дело с идентификатором, который компонент получает с URL-параметром `key`:

```
import { useState, useEffect } from 'react';
import { useParams } from 'react-router-dom';
import { getTodo } from './api.js';

export default function TodoDetail() {
    const [todo, setTodo] = useState([]);
    const { key } = useParams();

    useEffect(() => {
        (async () => {
            setTodo(await getTodo(key));
        })();
    }, [key]);
    . . .
}
```

Загрузку дела производит асинхронная функция `getTodo()` из модуля src\api.js. Загруженное дело сохраняется в созданном ранее состоянии `todo`.

Для извлечения значений URL-параметров, полученных компонентом, мы применили хук `useParams()` из модуля `react-router-dom` (подробности — в *главе 11*).

В вызове хука `useEffect()` мы указали массив зависимостей из одного элемента — переменной `key`, хранящей значение одноименного URL-параметра. Благодаря этому, эффект при последующих обновлениях компонента будет вызван лишь в том случае, если изменится значение этой переменной.

Функция, реализующая эффект, указанная в вызове хука, может вернуть результат — другую функцию, которая будет выполнена:

- при обновлении компонента — *перед* выполнением самого эффекта;
- при размонтировании компонента — в обязательном порядке.

Как правило, возвращаемая из эффекта функция производит отмену действий, выполняемых в эффекте (например, отмену привязки обработчика к событию). Вследствие чего ее называют *функцией очистки*.

Пример эффекта, привязывающего обработчик к событию `focus` окна и возвращающего функцию очистки, которая отменяет привязку обработчика к этому событию:

```
export default function SomeComponent() {
    . . .
    const activated = evt => { . . . };

    useEffect(() => {
        window.addEventListener('focus', activated);
        return () => {
            window.removeEventListener('focus', activated);
        };
    });
    . . .
}
```

Пример эффекта, устанавливающего соединение по протоколу WebSocket и возвращающего функцию очистки, которая разрывает это соединение:

```
useEffect(() => {
    const connection = new WebSocket('ws://somesite.ru:2345');
    connection.send('HELLO');
    connection.send(userId);
    connection.send(roomId);
    return () => {connection.close()};
}, [userId, roomId]);
```

Обратим внимание, что переменные `userId` и `roomId`, хранящие идентификаторы посетителя и комнаты чата, которые используются в теле эффекта, приведены

в массиве зависимостей. Вследствие чего при очередном обновлении компонента эффект будет вызван только лишь, если значение какой-либо из этих переменных изменится.

Эффекты, вызываемые перед рендерингом компонента, рекомендуется применять, если в теле эффекта не требуется обращаться к элементам содержимого этого компонента. Такие эффекты используются в большинстве случаев.

Если же в теле эффекта необходимо обратиться к какому-либо элементу содержимого компонента, чтобы, например, получить его размеры, следует оформить этот эффект как выполняемый после рендеринга компонента — когда все элементы его содержимого уже созданы.

9.1.2. Эффекты, вызываемые после рендеринга компонента

Эффект, вызываемый после рендеринга компонента, но перед выводом его на экран, создается вызовом хука `useLayoutEffect()` из модуля `react`. Он вызывается так же, как хук `useEffect()` (см. *разд. 9.1.1*).

Вот пример эффекта, получающего высоту компонента:

```
import { useState, useRef, useLayoutEffect } from 'react';

export default function Tooltip() {
    const compRef = useRef(undefined);

    const [tooltipHeight, setTooltipHeight] = useState(0);

    useLayoutEffect(() => {
        const { height } = ref.current.getBoundingClientRect();
        setTooltipHeight(height);
    }, []);
    . . .
    return (
        <div ref={compRef} style={{ height: tooltipHeight }}>
            . . .
        </div>
    );
}
```

Эффекты подобного рода применяются, если из тела эффекта необходимо получить доступ к какому-либо из элементов содержимого компонента.

> ***Внимание!***
>
> Следует помнить, что такие эффекты могут снизить производительность. Поэтому применять их следует лишь при крайней необходимости.

9.1.3. Эффекты, служащие для взаимодействия с внешним хранилищем данных

Начиная с React 18, во фреймворке присутствует поддержка еще одного вида эффектов — служащих для взаимодействия с внешним хранилищем данных. В качестве последнего может выступать как бэкенд, так и какая-либо структура, реализованная на самом JavaScript. Важно лишь то, что хранилище должно быть способно информировать компонент об изменении содержащихся в нем данных.

Эффект такого рода создается вызовом хука `useSyncExternalStore()` из модуля `react`:

```
useSyncExternalStore(<функция, регистрирующая обработчик изменения данных>,
                     <функция, выдающая данные>)
```

Первая из передаваемых хуку *функций* должна принимать в качестве параметра функцию-обработчик, которая будет вызываться при изменении данных, содержащихся в хранилище. Возвращать она должна функцию очистки, отменяющую привязку ранее заданной функции-обработчика.

Вторая из передаваемых хуку *функций* должна возвращать данные, содержащиеся в хранилище. Параметров она принимать не должна.

Обе *функции* должны быть синхронными.

Хук `useSyncExternalStore()` возвращает данные, содержащиеся в хранилище, которые выдаются второй *функцией*. Эти данные являются реактивными — при изменении содержимого хранилища они обновляются, что вызывает обновление компонента.

Хук работает следующим образом:

- при монтировании компонента — регистрирует обработчик, вызывая первую *функцию*, после чего вызывает вторую *функцию*, сохраняет возвращенные ею данные из хранилища и сам возвращает их в качестве результата;
- при обновлении компонента — возвращает ранее сохраненные данные из хранилища;
- при изменении содержимого хранилища — повторно вызывает вторую *функцию*, сохраняет выданные ею данные и возвращает их, инициируя обновление компонента;
- при размонтировании — вызывает функцию очистки, возвращенную первой *функцией*.

Вот пример эффекта, взаимодействующего с подсистемой разграничения доступа Google Firebase и выдающего сведения о текущем посетителе:

```
import { useSyncExternalStore } from 'react';
import { setStateChangeHandler, getCurrentUser } from './api.js';
export default function App() {
    . . .
```

```
    const user = useSyncExternalStore(
        setStateChangeHandler,
        getCurrentUser
    );
    . . .
}
```

Для привязки обработчика изменения статуса посетителя мы использовали функцию `setStateChangeHandler()` из модуля src\api.js, написанную при выполнении упражнения из *разд. 2.2*. Для получения текущего посетителя применяется функция `getCurrentUser()`, которую вы, уважаемые читатели, сами можете написать, взяв за основу функцию `getUserId()`, существующую в модуле src\api.js.

К сожалению, пока еще хук `useSyncExternalStore()` может правильно обрабатывать только синхронные функции. Если же указать в его вызове асинхронную функцию, хук не сможет извлечь нагрузку из промиса, возвращенного этой функцией.

9.2. Поддержка асинхронных операций

Многие операции, выполняемые с привлечением внешних API, являются асинхронными. В самую первую очередь это относится к операциям взаимодействия с бэкендом.

React предусматривает ряд весьма полезных инструментов, которые могут пригодиться при работе с асинхронными API.

9.2.1. Приостановщик

Приостановщик — программный механизм React, приостанавливающий вывод на экран своего содержимого до тех пор, пока не будет получен результат асинхронной операции.

Приостановщик реализуется компонентом `Suspense`, объявленным в модуле `react`. В него помещается содержимое, вывод которого необходимо приостановить до завершения асинхронной операции. А в его пропе `fallback` записывается JSX-код содержимого, которое станет выводиться на экран *во время* выполнения асинхронной операции.

Пример:

```
import { Suspense } from 'react';
. . .
<Suspense fallback={<p>Идет загрузка...</p>}>
    <SuspendedComponent />
</Suspence>
```

Наиболее часто реализуемый сценарий использования приостановщика таков:

1. В компоненте вызывается асинхронная функция, результат которой будет использоваться одним из компонентов-потомков, *без применения оператора* `await` (чтобы получить промис, а не его нагрузку).

2. Промис, возвращенный вызванной функцией, сохраняется в какой-либо переменной.
3. Потомок, использующий результат вызванной функции, заключается в приостановщик.
4. Промис, возвращенный вызванной функцией и сохраненный ранее, передается потомку через какой-либо проп.
5. Потомок в своем теле использует функцию `use()` из модуля `react` (поддерживается, начиная с React 19), чтобы извлечь нагрузку из полученного промиса:

   `use(<промис>)`

   Пока полученный промис не подтвержден, компонент-потомок не выводится на экран — вместо него там присутствует содержимое, заданное в пропе `fallback` приостановщика. Как только промис подтвердится, потомок будет выведен на экран.

Вот пример вывода сведений о выбранном деле с применением приостановщика:

```
import { Suspense, use } from 'react';
import { useParams } from 'react-router-dom';
import { getTodo } from './api.js';

export default function TodoDetail() {
    const { key } = useParams();

    const pTodo = getTodo(key);

    return (
        <Suspense fallback={<p>Идет загрузка...</p>}>
            <__TodoDetail pTodo={pTodo} />
        </Suspense>
    );
}

function __TodoDetail({ pTodo }) {
    const todo = use(pTodo);
    return ( . . . );
}
```

Для загрузки дела также используем асинхронную функцию `getTodo()` из модуля src\api.js. Только вызываем ее без оператора `await`, чтобы получить промис.

Для вывода собственно дела применяем вспомогательный компонент `__TodoDetail`, который заключаем в приостановщик и которому передаем ранее полученный промис — через проп `pTodo`. В теле вспомогательного компонента используем функцию `use()`, чтобы извлечь из промиса нагрузку.

Содержимое приостановщика может быть сколь угодно сложным — и всё оно на время выполнения асинхронной операции будет скрыто:

```
const pList = getTodos();
. . .
<Suspense fallback={<p>Идет загрузка...</p>}>
    <Layout />
        <LeftPanel />
        <Decoration>
            <TodoList pList={pList} />
        </Decoration>
        <Copyrignts />
    </Layout>
</Suspense>
```

Приостановщики можно вкладывать друг в друга. При этом будет задействован приостановщик, наиболее «близкий» к приостанавливаемому компоненту. Пример:

```
const pUser = getCurrentUser();
const pList = getTodos();
. . .
<Suspense fallback={<UserLoadIndicator />}>
    <Layout pUser={pUser} />
        <LeftPanel />
        <Decoration>
            <Suspense fallback={<TodoListLoadIndicator />}>
                <TodoList pList={pList} />
            </Suspense>
        </Decoration>
        <Copyrights />
    </Layout>
</Suspense>
```

9.2.2. Низкоприоритетные обновления компонента

Предположим, что мы реализовали в учебном приложении, написанном в *главах 1* и *2*, фильтрацию дел по вводимому искомому слову, причем непосредственно в процессе его ввода. И столкнулись с тем, что перечень дел обновляется каждый раз после ввода очередной буквы искомого слова. А, поскольку асинхронная загрузка списка дел с бэкенда и его вывод отнимают довольно много времени, приложение при этом сильно «тормозит».

Решить эту проблему можно, оформив обновление списка отфильтрованных дел как низкоприоритетное.

Низкоприоритетное обновление отличается от обычного тем, что, во-первых, выполняется, когда веб-обозреватель входит в состояние простоя, а во-вторых, не блокирует посетителю доступ к странице (вследствие чего он в процессе обновления может прокручивать содержимое, щелкать на гиперссылках, нажимать кнопки и др.). Все это повышает отзывчивость приложения.

Низкоприоритетные обновления появились в React 18. Они бывают двух видов.

9.2.2.1. Упрощенные низкоприоритетные обновления

Упрощенное низкоприоритетное обновление реализуется проще, но не позволяет выяснить текущее состояние обновления (выполнено оно или еще нет).

Упрощенное низкоприоритетное обновление задается вызовом функции `startTransition()` из модуля `react`:

```
startTransition(<функция, инициирующая низкоприоритетное обновление>)
```

Задаваемая *функция* не должна принимать параметров и возвращать результат. В ее теле следует выполнять изменение состояния, которое должно инициировать низкоприоритетное обновление. *Функция* будет выполнена, когда веб-обозреватель войдет в состояние простоя.

Результата функция `startTransition()` не возвращает.

Вот пример выводящего перечень дел компонента, позволяющего фильтровать дела по заданному искомому слову и использующего для этого упрощенное низкоприоритетное обновление:

```
import { useState, useEffect, startTransition } from 'react';
import { getTodos } from './api.js';

export default function TodoList() {
    const [list, setList] = useState([]);
    const [keyword, setKeyword] = useState(''); // ..................  1
    const [search, setSearch] = useState(''); // ....................  2

    useEffect(() => {
        (async () => {
            setList(await getTodos(search));
        })();
    });

    const handleKeywordChange = evt => {
        setKeyword(evt.target.value); // ............................  3
        startTransition(
            () => {setSearch(evt.target.value)} // ..................  4
        );
    };

    return (
        <section>
            <form>
                <input placeholder="Искомое слово"
                       value={keyword} onChange={handleKeywordChange} />
            </form>
            <table>
                {list.map(item => (
```

```
                    <tr key={item.key}>
                        . . .
                    </tr>
                ))}
            </table>
        </section>
    );
}
```

Для хранения заносимого посетителем искомого слова используем два состояния: `keyword`, которое применяется при обработке поля ввода в стиле React и изменение которого вызывает обычное обновление (поз. 1 в приведенном примере), и `search`, которое используется для фильтрации дел и изменение которого вызывает низкоприоритетное обновление (поз. 2).

В обработчике события `change` поля ввода искомого слова сначала заносим введенное слово в состояние `keyword`, чтобы вызвать обычное обновление, которое выведет это слово в поле ввода (поз. 3). Далее в вызове функции `startTransition()` заносим то же введенное слово в состояние `search` — это вызовет низкоприоритетное обновление, загружающее и выводящее новый список дел (поз. 4).

Низкоприоритетные обновления можно использовать совместно с приостановщиками (см. *разд. 9.2.1*);

```
import { useState, startTransition, Suspense, use } from 'react';
import { getTodos } from './api.js';

export default function TodoList() {
    const [keyword, setKeyword] = useState('');
    const [search, setSearch] = useState('');

    const pList = getTodos(search);

    const handleKeywordChange = evt => {
        setKeyword(evt.target.value);
        startTransition(
            () => {setSearch(evt.target.value)}
        );
    };

    return (
        <section>
            <form>
                <input placeholder="Искомое слово"
                       value={keyword} onChange={handleKeywordChange} />
            </form>
            <Suspense fallback={<p>Идет загрузка...</p>}>
                <__TodoList pList={pList} />
            </Suspense>
```

```
        </section>
    );
}

function __TodoList({ pList }) {
    const list = use(pList);
    . . .
    return ( . . . );
}
```

Упрощенные низкоприоритетные обновления рекомендуется применять везде, где не требуется отслеживать состояние обновления. Если же необходимо отслеживать состояние обновления, следует обратить внимание на полнофункциональные низкоприоритетные обновления.

9.2.2.2. Полнофункциональные низкоприоритетные обновления

Полнофункциональное низкоприоритетное обновление, в отличие от упрощенного, позволяет выяснить текущее состояние обновления (выполнено оно или еще нет), но реализуется несколько сложнее.

Полнофункциональное низкоприоритетное обновление задается с помощью хука `useTransition()` из модуля `react`, вызываемого без параметров.

Хук возвращает массив из двух элементов:

- `true`, если обновление в текущий момент выполняется, или `false`, если оно уже выполнено;
- функции, собственно задающей обновление и аналогичной функции `startTransition()` (см. *разд. 9.2.2.1*).

Как правило, возвращаемый массив подвергается деструктурированию. Переменной, в которой сохраняется первый элемент этого массива, обычно дается имя `isPending`, переменной для хранения второго элемента массива — `startTransition`.

Вот пример компонента, аналогичного рассмотренному в *разд. 9.2.2.1*, только использующего полнофункциональное низкоприоритетное обновление:

```
import { useState, useTransition, useEffect } from 'react';
import { getTodos } from './api.js';

export default function TodoList() {
    const [list, setList] = useState([]);
    const [keyword, setKeyword] = useState('');
    const [search, setSearch] = useState('');

    const [isPending, startTransition] = useTransition();
```

```
    useEffect(() => {
        (async () => {
            setList(await getTodos(search));
        })();
    });

    const handleKeywordChange = evt => {
        setKeyword(evt.target.value);
        startTransition(
            () => {setSearch(evt.target.value)}
        );
    };

    return (
        <section style={{ opacity: (isPending ? .5 : 1) }}>
            . . .
        </section>
    );
}
```

Полнофункциональные низкоприоритетные обновления также можно использовать совместно с приостановщиками.

Однако к полнофункциональным низкоприоритетным обновлениям имеет смысл прибегать лишь в тех случаях, когда необходимо отслеживать состояние обновления. Если состояние обновления не представляет интереса, лучше применять упрощенные низкоприоритетные обновления (см. *разд. 9.2.2.1*).

9.2.3. Величины с отложенным изменением

Величина с отложенным изменением — это реактивное значение, все изменения которого, выполняемые в коде компонента (и, соответственно, обновление этого компонента), откладываются до времени, когда веб-обозреватель начинает простаивать. Поддержка таких величин появилась в React 18.

Величина с отложенным изменением может принадлежать любому типу. Количество величин, создаваемых в компоненте, не ограничено.

Величина с отложенным изменением создается хуком `useDeferredValue()` из модуля `react`:

```
useDeferredValue(<величина, обновления которого необходимо задержать>[,
                 <изначальное значение этой величины>])
```

Поддержка второго параметра появилась в React 19.

Хук `useDeferredValue()` возвращает:

- при монтировании компонента — заданное вторым параметром *изначальное значение* или, если оно не указано, заданную первым параметром *величину*;

- при обновлении компонента — предыдущее значение указанной *величины*, *до* всех сделанных в коде компонента изменений;
- когда веб-обозреватель простаивает — актуальное значение указанной *величины*, *после* всех сделанных в коде компонента изменений.

  Как видим, все сделанные в компоненте изменения будут применены к заданной *величине* не сразу же, а как только веб-обозреватель начнет простаивать, что вызовет обновление компонента.

При размонтировании компонента величина с отложенным изменением удаляется из памяти.

Вот пример компонента, аналогичного рассмотренному в *разд. 9.2.2.1*, только хранящего искомое слово в виде величины с отложенным изменением:

```
import { useState, useDeferredValue, useEffect } from 'react';
import { getTodos } from './api.js';

export default function TodoList() {
    const [list, setList] = useState([]);
    const [keyword, setKeyword] = useState('');
    const search = useDeferredValue(keyword);

    useEffect(() => {
        (async () => {
            setList(await getTodos(search));
        })();
    });

    return (
        <section>
            <form>
                <input placeholder="Искомое слово"
                       value={keyword}
                       onChange={e => {setKeyword(e.target.value)}} />
            </form>
            . . .
        </section>
    );
}
```

Величины с отложенным изменением применяются в тех же случаях, что и низкоприоритетные обновления (см. *разд. 9.2.2*).

9.2.4. Оптимистичное обновление компонентов

Если бэкенд отлажен, ошибки при взаимодействии с ним маловероятны. И если мы отправим ему новое дело, созданное посетителем, то можем быть на 99,999% уверены, что это дело будет сохранено в базе. Так что, не дожидаясь, пока бэкенд

пришлет ответ с оповещением об успешном добавлении дела в базу (что может занять пару секунд), мы можем сразу же вывести новое дело в перечне.

То есть произвести *оптимистичное обновление* компонента — обновление, выполняемое в твердой уверенности, что операция с бэкендом завершилась успехом. Поддержка оптимистичных обновлений появилась в React 19.

Оптимистичное обновление реализуется с помощью хука `useOptimistic()` из модуля `react`:

```
useOptimistic(<текущее состояние>, <функция оптимистичного обновления>)
```

Функция оптимистичного обновления должна принимать два параметра: текущее состояние и пользовательские данные, которые следует добавить к копии текущего состояния, чтобы получить новое состояние — с внесенным оптимистичным обновлением. Пользовательские данные могут быть любого типа. В качестве результата *функция* должна возвращать полученное новое состояние.

Хук `useOptimistic()` возвращает массив из двух элементов:

- актуальное состояние:
  - до оптимистичного обновления — равное *текущему*, заданному в вызове хука;
  - после оптимистичного обновления — копию текущего состояния с внесенными в него пользовательскими данными.

    Поскольку актуальное состояние относится к реактивным данным, следует возвращать именно копию текущего состояния, в которую внесены пользовательские данные. Только в этом случае компонент будет обновлен;
- функция, обновляющее состояние. Она должна принимать с единственным параметром пользовательские данные, добавляемые к состоянию, и не должна возвращать результат.

Вот пример компонента, выводящего отдельную позицию перечня дел, позволяющего помечать дело как выполненное и использующего для этого оптимистичное обновление:

```
import { useOptimistic } from 'react';
import { useSubmit } from 'react-router-dom';

export default function TodoItem({ item: __item }) {
    const [item, setItemDone] = useOptimistic(
        __item,
        (currentItem, optData) => {
            return { ...currentItem, done: optData }; // .............  2
        }
    );
    const submit = useSubmit();

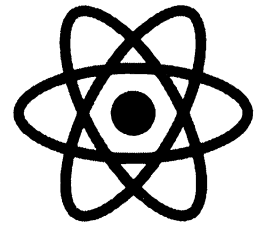
    const handleDoneClick = () => {
        setItemDone(true); // ....................................  1
```

```
        submit(null, {action: `/${item.key}`, method: 'patch'});
    };

    return (
        <tr>
            <td>
                {item.done && <del>{item.title}</del>}
                {!item.done && item.title}
            </td>
            <td>
                <button title="Выполнено" disabled={item.done}
                        onClick={handleDoneClick}>
                    &#9745;
                </button>
            </td>
        </tr>
    );
}
```

Код здесь не оптимален: значение `true`, заносимое в свойство `done` дела, передается в параметре функции, обновляющей состояние (поз. 1 и 2 в приведенном примере), тогда как его можно было бы записать непосредственно в теле функции оптимистичного обновления, передаваемой хуку `useOptimistic()` вторым параметром. Однако в качестве простого примера он вполне приемлем.

ГЛАВА 10



Компоненты высшего порядка и собственные хуки

10.1. Компоненты высшего порядка

Компонент высшего порядка — это функция, которая принимает в качестве параметров компонент и некоторые настройки и возвращает другой компонент — с расширенной функциональностью. Последний включает изначальный компонент, измененный соответственно полученным настройкам, и, возможно, дополнительную HTML-разметку.

Процесс расширения функциональности изначального компонента путем оснащения его дополнительной разметкой и программной логикой называется *композицией*.

10.1.1. Компонент высшего порядка, пример 1

Предположим, что у нас есть компонент `Control`, реализующий элемент управления HTML. Для простоты в нем предусмотрено создание лишь тех элементов управления, которые формируются тегами `<input>` и `<textarea>`. Этот компонент принимает следующие пропы:

- `type` — обозначение типа компонента: либо одно из значений, поддерживаемых атрибутом тега `type`, либо `'textarea'` для создания области редактирования (по умолчанию: `'text'`);
- `onValue` — функция, которая принимает в качестве параметра значение, занесенное в элемент управления, и выполняет с ним заданные действия (например, заносит в состояние компонента-родителя);
- любые произвольные пропы, которые будут переданы элементу управления без изменений.

Код этого компонента приведен в листинге 10.1.

Листинг 10.1. Компонент `Control`

```
import { useState } from 'react';

export default function Control({ type='text', onValue, ...others }) {
    const [value, setValue] = useState('');
```

```
    const handleInputChange = evt => {
        setValue(evt.target.value);
        onValue(evt.target.value);
    };

    if (type === 'textarea')
        return (<textarea {...others} value={value}
                          onChange={handleInputChange} />);
    else
        return (<input type={type} {...others} value={value}
                       onChange={handleInputChange} />);
}
```

Пример использования такого компонента:

```
<label>Заголовок</label>
<Control className="input" onValue={setTitle} />
<label>Примечание</label>
<Control type="textarea" className="textarea" onValue={setDesc} />
```

Теперь нам нужно с помощью этого компонента создать элементы управления, оформленные в стиле Bulma (с надписями), и реализовать у этих компонентов вывод сообщений об ошибках ввода. Но у нас нет ни времени, ни желания писать необходимую HTML-разметку для каждого компонента.

Выход — объявление компонента высшего порядка с именем, например, `withBulmaLayout`, который сделает за нас всю работу:

```
withBulmaLayout(<изначальный компонент>, <текст надписи>)
```

Компонент, возвращенный этой функцией, будет поддерживать дополнительный проп `errorMessage`, задающий сообщение об ошибке ввода.

Код компонента высшего порядка `withBulmaLayout` приведен в листинге 10.2.

Листинг 10.2. Компонент высшего порядка `withBulmaLayout`

```
export function withBulmaLayout(Comp, label) {
    return ({ type, className='', errorMessage='', ...others }) => {
        let cls = (type === 'textarea') ? 'textarea' : 'input';
        if (!className)
            cls += ' ' + className;

        return (
            <div className="field">
                <label className="label">{label}</label>
                <div className="control">
                    <Comp type={type} className={cls} {...others} />
                </div>
```

```
                {errorMessage &&
                    <p className="help is-danger">
                        {errorMessage}
                    </p>
                }
            </div>
        );
    }
}
```

Пример использования этого компонента:

```
const BulmaTitle = withBulmaLayout(Control, 'Заголовок');
const BulmaDesc = withBulmaLayout(Control, 'Примечание');
. . .
return (
    . . .
    <BulmaTitle onValue={setTitle} />
    <BulmaDesc type="textarea" onValue={setDesc} />
    . . .
);
```

10.1.2. Компонент высшего порядка, пример 2

Предположим, мы написали два компонента. Компонент `TodoList` загружает список дел с бэкенда и выводит его на странице:

```
import { useState, useEffect } from 'react';
import { getTodos } from './api.js';

export default function TodoList() {
    const [data, setData] = useState([]);

    useEffect(() => {
        (async () => {
            setData(await getTodos());
        })();
    });

    return (
        <table>
            {data.map(item => (
                . . .
            ))}
        </table>
    );
}
```

А компонент `TodoDetail` загружает с бэкенда и выводит дело с указанным в URL-параметре `key` идентификатором:

```
import { useState, useEffect } from 'react';
import { useParams } from 'react-router-dom';
import { getTodo } from './api.js';

export default function TodoDetail() {
    const [data, setData] = useState([]);
    const { key } = useParams();

    useEffect(() => {
        (async () => {
            setData(await getTodo(key));
        })();
    }, [key]);

    return (
        <section>
            <h1>{data.title}</h1>
            . . .
        </section>
    );
}
```

Как можно видеть, значительная часть кода обоих компонентов практически одинакова. Поэтому ее стоит вынести в компонент высшего порядка `withLoadData`, а в компонентах `TodoList` и `TodoDetail` оставить лишь код, формирующий содержимое.

Формат вызова функции компонента `withLoadData`:

```
withLoadData(<изначальный компонент>, <функция, загружающая данные>)
```

Задаваемая *функция* должна принимать в качестве параметра объект с URL-параметрами и возвращать промис, содержащий загруженные данные.

Код компонента `withLoadData` показан в листинге 10.3.

Листинг 10.3. Компонент высшего порядка `withLoadData`

```
import { useState, useEffect } from 'react';
import { useParams } from 'react-router-dom';

export default function withLoadData(Comp, loadFn) {
    return props => {
        const [data, setData] = useState([]);
        const urlParams = useParams();
```

```
        useEffect(() => {
            (async () => {
                setData(await loadFn(urlParams));
            })();
        });

        return <Comp data={data} {...props} />;
    };
}
```

Примеры использования компонента `withLoadData`:

```
// Модуль src\TodoList.js
function __TodoList({ data }) {
    return (
        <table>
            {data.map(item => (
                . . .
            ))}
        </table>
    );
}
const TodoList = withLoadData(__TodoList, async () => getTodos());
export default TodoList;

// Модуль src\TodoDetail.js
function __TodoDetail({ data }) {
    return (
        <section>
            <h1>{data.title}</h1>
            . . .
        </section>
    );
}
const TodoDetail = withLoadData(__TodoDetail,
                                async urlParams => getTodo(urlParams.key));
export default TodoDetail;

// Модуль src\App.js
return (
    . . .
    <TodoList className="decorated" />
    <TodoDetail ref={detailRef} />
    . . .
);
```

10.2. Написание собственных хуков

Хук React реализуется в виде функции, которая может принимать произвольное количество параметров (в том числе и ни одного) и возвращать или не возвращать результат. Единственное требование — в теле хука должен находиться хотя бы один вызов другого хука (неважно, реализованного в какой-либо из библиотек или написанного разработчиком приложения).

По существующему соглашению имя хука должно начинаться с префикса `use`.

Для примера напишем хук `useLoadData()`, загружающий и выдающий данные с бэкенда. Формат его вызова будет таким:

```
useLoadData(<функция, загружающая данные>[, <изначальное значение>=[]])
```

Задаваемая *функция* должна принимать в качестве параметра объект с URL-параметрами и возвращать промис, содержащий загруженные данные.

Хук будет возвращать:

- если данные еще не загружены — указанное *изначальное значение* (по умолчанию: пустой массив);
- если данные уже загружены — промис с загруженными данными.

Код хука `useLoadData()`:

```
import { useState, useEffect } from 'react';
import { useParams } from 'react-router-dom';

export function useLoadData(loadFn, initial=[]) {
    const [data, setData] = useState(initial);
    const urlParams = useParams();

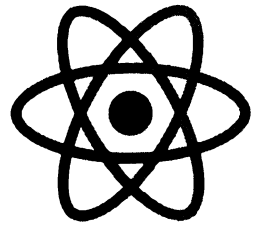
    useEffect(() => {
        (async () => {
            setData(await loadFn(urlParams));
        })();
    });

    return data;
}
```

Примеры использования этого хука:

```
// Модуль src\TodoList.js
export default function TodoList() {
    const list = useLoadData(async () => getTodos());
    . . .
}

// Модуль src\TodoDetail.js
export default function TodoDetail() {
    const todo = useLoadData(async urlParams => getTodo(urlParams.key));
    . . .
}
```



ЧАСТЬ III

Дополнительные библиотеки и расширенные инструменты

Глава 11. Библиотека React Router: навигация, раздел 1

Глава 12. Библиотека React Router: навигация, раздел 2

Глава 13. Преобразователи данных React

Глава 14. Библиотека Redux: централизованное хранение состояния, раздел 1

Глава 15. Библиотека Redux: централизованное хранение состояния, раздел 2

Глава 16. Библиотека Formik: создание веб-форм

Глава 17. Библиотека Yup: валидация данных

Глава 18. Библиотека ReactFire: интеграция с Google Firebase

Глава 19. Библиотека React Awesome Reveal: анимация

Глава 20. Прочие полезные инструменты и библиотеки

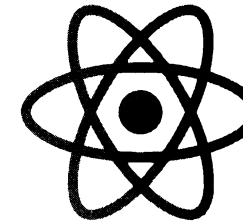
Глава 21. Обработка ошибок

Глава 22. Разделение кода и загрузка по запросу

Глава 23. Расширенные средства отладки

Глава 24. Публикация веб-приложения

ГЛАВА 11



Библиотека React Router: навигация, раздел 1

Интерфейс большинства веб-приложений весьма сложен, вследствие чего его разделяют на ряд отдельных экранов. Например, приложение списка планируемых дел, написанное в *главах 1* и *2*, содержит пять экранов: перечня дел, сведений о выбранном деле, создания нового дела, регистрации нового посетителя и входа в приложение. Посетитель в процессе работы с приложением перемещается с одного экрана на другой.

В простейших случаях для перемещения между экранами можно предусмотреть специальные кнопки. Но гораздо удобнее реализовать полноценную *навигацию*, при которой каждому экрану приложения ставится в соответствие определенный интернет-адрес. Это дает два преимущества:

- для перехода с экрана на экран можно использовать обычные гиперссылки;
- между ранее посещенными экранами можно перемещаться, нажимая кнопки **Назад** и **Вперед**.

Создавать навигацию в React-приложении лучше всего с помощью дополнительной библиотеки React Router. Установить ее можно подачей команды:

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

> ***НА ЗАМЕТКУ***
>
> Полное описание библиотеки React Router находится по интернет-адресу: **https://reactrouter.com/**.

11.1. Основные принципы и понятия React Router

1. При навигации по одностраничному веб-приложению (а React-приложение таким и является) в интернет-адресе изменяется лишь путь.

   Путь — часть интернет-адреса, располагающаяся после адреса хоста и номера TCP-порта (если он присутствует) и перед набором GET-параметров (опять же, если они присутствуют). В традиционных многостраничных сайтах путь указывает на файл, который требуется загрузить с сайта.

Путь состоит из отдельных *сегментов*, разделенных слешами.

Например, в интернет-адресе **http://someapp.ru/todos/1627298956192** путь — **/todos/1627298956192**. Он состоит из сегментов: **todos** и **1627298956192**.

Путь / (прямой слеш, «корень» приложения) будет вести на *главный экран* приложения — эквивалент главной страницы традиционного сайта.

2. Каждый из имеющихся в приложении компонентов-экранов следует связать с определенным путем, называемым *шаблонным*. При переходе по этому пути связанный с ним компонент (называемый *целевым*) будет выведен на экран. Шаблонный путь для каждого целевого компонента можно выбрать произвольно.

   Например, для перехода на компонент-экран регистрации можно выбрать шаблонный путь **/register**, а для перехода на экран сведений о приложении — шаблонный путь **/about**.

3. Для каждого шаблонного пути создается отдельный маршрут.

   Маршрут — компонент, связывающий какой-либо целевой компонент-экран с выбранным для него шаблонным путем.

4. В составе пути целевому компоненту можно передать набор произвольных значений, называемых *URL-параметрами*. Целевой компонент извлечет их и использует в работе. URL-параметры записываются в шаблонном пути с применением особых обозначений.

   Так, в составе приведенного ранее пути сегмент **1627298956192** — это URL-параметр, посредством которого передается идентификатор подлежащего выводу дела.

   Маршруты, содержащие URL-параметры в шаблонных путях, называются *параметризованными*.

5. Все маршруты сводятся в *список маршрутов*.

6. На основе полученного списка маршрутов создается маршрутизатор.

   Маршрутизатор — подсистема, отслеживающая изменение интернет-адреса (при переходе по гиперссылкам или выполненное программно), находящая маршрут, шаблонный путь которого совпадает с путем, извлеченным из текущего интернет-адреса (*текущим путем*), т. е. *совпавший* маршрут, и выводящая записанный в нем целевой компонент на экран.

   Процесс нахождения совпавшего маршрута и вывода соответствующего целевого компонента называется *маршрутизацией*.

7. Маршрутизатор помещается на страницу с помощью особого компонента, называемого *провайдером маршрутизации*.

   Как правило, провайдер маршрутизации указывается в содержимом, выводимом в стартовом модуле, вместо компонента приложения.

8. В других компонентах создаются гиперссылки, указывающие на различные экраны приложения и содержащие соответствующие им пути.

9. При необходимости можно выполнить *перенаправление* — принудительный программный переход по другому интернет-адресу.

> **Внимание!**
> Компоненты маршрута, двух видов гиперссылок, перенаправления, все хуки и функции, используемые для создания навигации, объявлены в модуле `react-router-dom`.

11.2. Маршруты, списки маршрутов и маршрутизаторы

11.2.1. Маршрут

Отдельный маршрут в React Router реализуется компонентом `Route`. Он принимает следующие пропы:

- `path` — шаблонный путь. В зависимости от разновидности списка маршрутов указывается либо с начальным слешем, либо без него (подробности — в *разд. 11.2.2.2*).

  Конечный слеш в шаблонном пути может как присутствовать, так и отсутствовать. Все равно он игнорируется маршрутизатором;

- `element` — JSX-код, создающий целевой компонент, который должен быть выведен на странице при совпадении текущего маршрута:

```
import { Route } from 'react-router-dom';
. . .
<Route path='/' element={<TodoList />} />
<Route path='/add' element={<TodoAdd />} />
```

  Можно указать более сложное содержимое, состоящее из любых HTML-тегов и React-компонентов:

```
<Route path='/about' element={(
    <section>
        <h1>О приложении</h1>
        <p>Приложение списка планируемых дел.</p>
        <p>Разработано читателями книги &laquo;React 19&raquo;.</p>
    </section>
)} />
```

  Также можно указать любое «пустое» значение: `undefined`, `null`, `true` или `false` — в этом случае при совпадении маршрута на страницу ничего выведено не будет:

```
<Route path='/empty' element={undefined} />
```

  Значение пропа по умолчанию: `null`;

- `Component` (с большой буквы!) — ссылка на функцию, реализующую целевой компонент:

```
<Route path='/' Component={TodoList} />
```

Применение этого пропа позволяет немного сократить код. Однако делать так можно лишь в случае, если целевому компоненту не требуется передавать никакие пропы.

> ***Внимание!***
>
> **Если в маршруте присутствуют пропы `element` и `Component`, будет принят во внимание только проп `Component`.**

- `caseSensitive` — если `true`, при сравнении текущего пути с шаблонным будет учитываться регистр символов, если `false` — не будет (по умолчанию: `false`):

```
<Route path='/ADD' element={<TodoAddSpecial />} caseSensitive />
<Route path='/add' element={<TodoAddOthers />} />
```

Еще несколько пропов, поддерживаемых компонентом `Route`, мы изучим позже.

Компонент `Route` может содержать другие маршруты. Этим пользуются при создании иерархического списка маршрутов (см. *разд. 11.2.2.2.2*).

11.2.1.1. Написание шаблонных путей. Запись URL-параметров

Шаблонный путь, записываемый в пропе `path` маршрута, может содержать в себе сегменты следующих форматов:

- простая строка — статичный сегмент:

Шаблонный путь	Совпадающий текущий путь
/add	**/add**
/user/register	**/user/register**

- `:<имя>` — URL-параметр с заданным *именем*. *Имя* должно быть уникальным в пределах шаблонного пути и удовлетворять требованиям, предъявляемым к именам переменных JavaScript. Примеры:

Шаблонный путь	Совпадающие текущие пути
/:key	**/123456789**, **/876da7wada6**, **/somewhere**
/user/:userId/todos/:key	**/user/vasya/todos/123456789**, **/user/superpuper/todos/megadeed**

Как показано в примерах, в составе пути может быть больше одного URL-параметра;

- `<сегмент>?` — необязательный *сегмент*. Может быть как статическим сегментом, так и URL-параметром. Примеры:

Шаблонный путь	Совпадающие текущие пути
/user?/login	**/user/login**, **/login**
/:userId?/add	**/vasya/add**, **/petya/add**, **/add**

- * (звездочка) — ставится в конце пути и обозначает все оставшиеся символы, включая слеши:

Шаблонный путь	Совпадающие текущие пути
/login/*	/login, /login/abcdefgh, /login/to/app
/:key/*	/123456789, /123456789/987654321/909

Шаблонный путь /* совпадает с любым текущим путем, который не совпал ни с одним из шаблонных путей, присутствующих во всех имеющихся маршрутах.

11.2.2. Список маршрутов

11.2.2.1. Создание списка маршрутов

Список маршрутов в формате, пригодном для формирования на его основе маршрутизатора, создается функцией `createRoutesFromElements()`:

```
createRoutesFromElements(<массив маршрутов>)
```

В качестве параметра задается массив маршрутов, которые войдут в создаваемый список. Эти маршруты должны быть представлены компонентами `Route`.

Функция возвращает объект, представляющий список маршрутов.

Пример:

```
import { Route, createRoutesFromElements } from 'react-router-dom';

const rm = [
    <Route path="/" element={<TodoList />} />,
    <Route path="/add" element={<TodoAdd />} />,
    <Route path="/:key" element={<TodoDetail />} />,
    <Route path="/login" element={<Login />} />,
    <Route path="/logout" element={<Logout />} />,
    <Route path="/register" element={<Register />} />
];

const rs = createRoutesFromElements(rm);
```

Как правило, список маршрутов создается в отдельном модуле и подвергается экспорту по умолчанию. Модулю со списком маршрутов обычно дают имя src\routes.js.

11.2.2.2. Две разновидности списка маршрутов

React Router позволяет создавать списки маршрутов, относящиеся к двум разновидностям. И, соответственно, реализовать две разновидности навигации.

11.2.2.2.1. Простой список маршрутов

Простой список маршрутов включает только маршруты, не содержащие потомков. Пример простого списка маршрутов приведен в *разд. 11.2.2.1*.

Особенности простого списка маршрутов:

1. Компоненты `Route`, создающие маршруты, не имеют содержимого.
2. Шаблонные пути всегда записываются с начальными слешами.
3. При навигации целевые компоненты выводятся непосредственно на странице, занимая ее полностью.

   Так, при переходе по пути / (слеш) на страницу будет выведен компонент `TodoList`, заняв ее полностью, а ранее присутствовавший там компонент будет удален. Если же посетитель перейдет по пути **/add**, то на странице появится компонент `TodoAdd`, а ранее выведенный компонент `TodoList` удалится.

Достоинство простого списка маршрутов — легкость написания и наглядность. Однако, если какие-либо элементы должны присутствовать на странице постоянно (к ним относятся шапка, панель навигации, поддон, элементы разметки и т. п.), их придется размещать в каждом компоненте-экране, что весьма трудоемко.

Простой список маршрутов имеет смысл использовать лишь при разработке очень простых приложений. В остальных случаях настоятельно рекомендуется создать иерархический список и реализовать иерархическую навигацию.

11.2.2.2.2. Иерархический список маршрутов

Иерархический список маршрутов содержит иерархию маршрутов, в которой одни маршруты вложены в другие.

Принципы, по которым создается такой список маршрутов, перечислены далее.

1. Как говорилось ранее, одни маршруты вкладываются в другие.

   Маршруты, вложенные в другой маршрут, называются *дочерними*, а маршрут, в который они вложены, — *родительским*.

   Компоненты `Route`, создающие дочерние маршруты, размещаются непосредственно в компоненте `Route` родительского маршрута.
2. В один родительский маршрут вкладываются все дочерние маршруты, у которых шаблонные пути начинаются с одинакового префикса. При этом в пропе `path` записывается:
   - у родительского маршрута — префикс шаблонного пути;
   - у дочерних маршрутов — шаблонные пути *без префикса*.

   Вложенность маршрутов друг в друга не ограничена.

   Результирующий шаблонный путь каждого маршрута составляется из «остатка» шаблонного пути, записанного непосредственно в нем, и всех префиксов, заданных во всех родительских маршрутах, в которые он последовательно вложен.
3. Один из дочерних маршрутов следует сделать *корневым* — имеющим тот же шаблонный путь, что и родительский маршрут, — одним из двух способов:
   - задав в нем в качестве значения пропа `path` пустую строку;
   - задав в нем проп `index` со значением `true`.

4. Маршруты, содержащие в себе дочерние маршруты, называются *узловыми*, не содержащие их — *конечными*.
5. Во всех маршрутах — и узловых, и конечных — указываются целевые компоненты.

В качестве примера рассмотрим следующий простой список маршрутов:

```
import { Route } from 'react-router-dom';
const rm = [
    <Route path="/" element={<TodoList />} />,
    <Route path="/add" element={<TodoAdd />} />,
    <Route path="/:key" element={<TodoDetail />} />,
    <Route path="/user" element={<Profile />} />,
    <Route path="/user/login" element={<Login />} />,
    <Route path="/user/logout" element={<Logout />} />,
    <Route path="/user/register" element={<Register />} />
];
```

Все шесть маршрутов содержат шаблонные пути, начинающиеся с одинакового префикса — / (слеш). Кроме того, у последних трех маршрутов шаблонные пути также начинаются с одинакового префикса — **user/** (без учета начального слеша).

Значит, этот список маршрутов можно преобразовать в иерархический, имеющий три уровня вложенности:

```
const rm2 = (
    <Route path="/" element={<App />}>
        <Route path="" element={<TodoList />} />
        <Route path="add" element={<TodoAdd />} />
        <Route path=":key" element={<TodoDetail />} />
        <Route path="user" element={<UserLayout />}>
            <Route index element={<Profile />} />
            <Route path="login" element={<Login />} />
            <Route path="logout" element={<Logout />} />
            <Route path="register" element={<Register />} />
        </Route>
    </Route>
);
```

Обратим внимание, что в узловых маршрутах записываются префиксы шаблонных путей, а в конечных маршрутах — сами пути без префиксов.

Результирующий путь маршрута, указывающего на компонент `TodoDetail`, — **/:key**, а результирующий путь маршрута, который ссылается на компонент `Logout`, — **/user/logout**.

Маршрут, указывающий на компонент `TodoList`, является корневым у своего родителя. Он помечен как корневой первым способом (см. п. 3) — указанием в нем пропа `path` с пустой строкой. Шаблонный путь этого маршрута — / (слеш) —

схож с таковым у его родителя. Соответственно, переход по пути / приведет к выводу компонента `TodoList`.

Маршрут, указывающий на компонент `Profile`, также является корневым у родителя. Он помечен как корневой вторым способом — указанием пропа `index` со значением `true` (как сказано в *разд. 5.1.1*, для задания в компоненте пропа с таким значением достаточно записать лишь сам проп). Шаблонный путь этого маршрута и его родителя — **/user**, при переходе по этому пути будет выведен компонент `Profile`.

6. Маршрутизатор «следует» от одного узлового маршрута к другому «вглубь» иерархии, пока не найдет совпавший маршрут.

   Так, при переходе по пути **/user/register** маршрутизатор проследует от узлового маршрута с префиксом / через узловой маршрут с префиксом **user** к конечному маршруту с шаблонным путем **register** и выведет компонент `Register`.

   В иерархическом списке маршрутов совпавшими могут оказаться сразу несколько маршрутов — конечный и все узловые, в которые он последовательно вложен.

7. *Все целевые компоненты*, записанные во всех — и узловых, и конечных — маршрутах, через которые «проследует» маршрутизатор, *будут выведены на экран*.

   Компонент из первого, наиболее «внешнего» узлового маршрута будет выведен непосредственно на странице, в нем отобразится компонент из узлового маршрута второго уровня вложенности, в нем — компонент из узлового маршрута третьего уровня вложенности, и т. д. Компонент из конечного маршрута отобразится внутри компонента из последнего узлового маршрута, в который он вложен.

   В результате получится иерархия вложенных друг в друга компонентов, аналогичная иерархии самих маршрутов.

   Например, при переходе по пути **/user/register** на странице будет выведен компонент `App`, в нем — компонент `UserLayout`, а в нем — компонент `Register`.

8. Место в компоненте, в котором должны выводиться компоненты из дочерних маршрутов, помечается особым компонентом, называемым *выпуском* (подробности — далее).

9. В целевом компоненте из наиболее «верхнего» узлового маршрута размещаются элементы, которые должны присутствовать на экране постоянно. В первую очередь это разметка (отчего компоненты из узловых маршрутов часто называют *разметочными*). Также в этом компоненте обычно присутствуют шапка, панель навигации, поддон и другие подобного рода элементы. В разметочных компонентах из вложенных узловых маршрутов присутствуют элементы, создающие дополнительную разметку, дополнительные панели навигации и т. п.

Иерархический список маршрутов создается несколько сложнее, чем простой. Однако его применение позволяет «облегчить» компоненты-экраны, перенеся из них все элементы, присутствующие на экране постоянно, в разметочные компоненты.

11.2.3. Выпуск

Если в приложении реализован иерархический список маршрутов, требуется указать место в коде каждого разметочного компонента, в котором должны выводиться компоненты из дочерних маршрутов. Это выполняется вставкой в JSX-код особой пометки, называемой *выпуском*[1].

Выпуск реализуется компонентом `Outlet`. Он всегда указывается без содержимого и не поддерживает никаких специфических пропов.

Пример:

```
import { Outlet } from 'react-router-dom';
export default function App() {
    . . .
    return (
        <main>
            <Outlet />
        </main>
    );
}
```

11.2.4. Маршрутизаторы

React Router предоставляет две разновидности маршрутизаторов.

11.2.4.1. Маршрутизатор Browser Router

Маршрутизатор *Browser Router* рекомендуется к применению во всех случаях, когда это не вызывает проблем (о них — далее).

Он создается на основе заданного *списка маршрутов* функцией `createBrowserRouter()`:

```
createBrowserRouter(<список маршрутов>[, <настройки>])
```

Список маршрутов может быть как простым (см. *разд. 11.2.2.2.1*), так и иерархическим (см. *разд. 11.2.2.2.2*).

Функция возвращает готовый маршрутизатор, представленный особым объектом.

Пример:

```
import { createRoutesFromElements, createBrowserRouter }
       from 'react-router-dom';
const rm = ( . . . );
const rs = createRoutesFromElements(rm);
const router = createBrowserRouter(rs);
```

В вызове функции `createBrowserRouter()` можно указать необязательные *настройки*. Они записываются в виде простого объекта, каждое свойство которого представля-

[1] По-английски — outlet.

ет одну настройку. В настоящее время представляет интерес настройка `basename`, которая задает префикс, добавляемый к путям из гиперссылок, создаваемых React Router. Префикс указывается в виде строки — с начальным слешем, но без конечного. Пример:

```
const router2 = createBrowserRouter(rs, { basename: '/todos' });
```

Маршрутизатор Browser Router создает гиперссылки с простыми и понятными путями, не содержащими специальных символов наподобие решеток (#).

Однако у этого маршрутизатора есть два существенных недостатка: первый — он не работает в старых веб-обозревателях (что, в принципе, не очень страшно, поскольку сейчас мало кто ими пользуется), однако второй недостаток серьезнее, и его лучше рассмотреть на примере.

Предположим, посетитель пытается перейти на экран добавления нового дела, задав его интернет-адрес: **http://todosapp.ru/items/add**. Веб-сервер, обслуживающий приложение, получит от веб-обозревателя клиентский запрос с путем **/items/add** и решит, что этот путь указывает на файл add из папки items, хранящейся в корневой папке. Такого файла там, скорее всего, нет, и веб-сервер отправит веб-обозревателю серверный ответ с кодом статуса 404 (запрошенный файл не найден).

Чтобы веб-сервер в подобных случаях не искал отсутствующий файл, а отправлял веб-обозревателю страницу приложения, его следует соответственно сконфигурировать (как — описано в *главе 24*). Если приложение публикуется на мощностях стороннего хостинг-провайдера, с этим могут быть проблемы.

Разработчики React Router рекомендуют везде, где возможно, применять Browser Router как порождающий более привычные пути. Если же в работе приложения возникают проблемы, связанные с навигацией, следует рассмотреть использование маршрутизатора Hash Router.

11.2.4.2. Маршрутизатор Hash Router

Маршрутизатор *Hash Router* позволяет избежать описанной в *разд. 11.2.4.1* проблемы с веб-серверами, принадлежащими сторонним хостинг-провайдерам.

Этот маршрутизатор помещает в начало пути, входящего в состав интернет-адреса каждой выводимой на странице гиперссылки, символ решетки #, тем самым фактически превращая путь в якорь. Так, если в гиперссылке был записан путь **/items/add**, то при выводе ее на страницу он будет преобразован в **#/items/add**. Веб-сервер, получив клиентский запрос с таким путем, уберет из него якорь **#/items/add** и получит путь / (слеш), указывающий на страницу приложения. Которую и отправит веб-обозревателю.

Маршрутизатор Hash Router создается функцией `createHashRouter()`, полностью аналогичной функции `createBrowserRouter()` (см. *разд. 11.2.4.1*).

Пример:

```
import { createHashRouter } from 'react-router-dom';
. . .
const router = createHashRouter(rs);
```

У такого маршрутизатора есть два преимущества:

- он хорошо работает в старых веб-обозревателях, не поддерживающих стандарт HTML 5;
- приложение, использующее Hash Router, может быть без проблем опубликовано на мощностях любого хостинг-провайдера.

Недостатком этого маршрутизатора является лишь то, что он добавляет в каждый путь лишний символ решетки #, а это может быть критично в каких-либо случаях.

11.2.5. Провайдер маршрутизации

Провайдер маршрутизации — это компонент, который, собственно, заставляет работать все инструменты React Router, используемые в приложении.

Компонент провайдера маршрутизации носит имя `RouterProvider` и поддерживает два следующих пропа:

- `router` — маршрутизатор, созданный ранее;
- `fallbackElement` — JSX-код содержимого, которое будет выводиться, пока провайдер маршрутизации монтируется. Если маршрутов в списке много, процесс монтирования может затянуться, и имеет смысл показать посетителю какое-либо сообщение о том, что приложение в текущий момент запускается. По умолчанию: `null`.

Практически всегда провайдер маршрутизации выводится в составе содержимого, отображаемого на странице в стартовом модуле src\index.js.

Вот пример отображения провайдера маршрутизации, использующего маршрутизатор из модуля src\routes.js и выводящего компонент `Spinner` во время монтирования:

```
import { RouterProvider } from 'react-router-dom';
import router from './routes.js';
import Spinner from './Spinner.js';

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(
    <React.StrictMode>
        <RouterProvider router={router}
                        fallbackElement={<Spinner size="big" />} />
    </React.StrictMode>
);
```

11.3. Навигация

React Router предоставляет две разновидности инструментов для выполнения навигации: гиперссылки и средства перенаправления.

11.3.1. Гиперссылки

React Router позволяет создавать гиперссылки двух видов: упрощенные и полнофункциональные.

11.3.1.1. Упрощенные гиперссылки

Упрощенная гиперссылка дает возможность лишь переходить на другой экран. Однако она не способна каким-либо образом сигнализировать посетителю о том, что указывает на текущий путь (является *активной*).

Упрощенная гиперссылка представляется компонентом `Link`. Он поддерживает следующие пропы:

- `to` — путь в виде:
  - строки:

```
import { Link } from 'react-router-dom';
. . .
<Link to="/add">Создать дело</Link>
```

    Путь может включать GET-параметры и якорь:

```
<Link to="/find?keyword=react#results">
    Вывести результаты поиска по слову «react»
</Link>
```

  - простого объекта со следующими свойствами, содержащими строковые значения:
    - `pathname` — собственно путь;
    - `search` — набор GET-параметров без начального символа `?`;
    - `hash` — якорь без начального символа решетки `#`.

    Пример:

```
const to = { pathname: '/find', search: 'keyword=react',
                                hash: 'results' };
. . .
<Link to={to}>
    Вывести результаты поиска по слову 'react'
</Link>
```

- `replace` — если `true`, текущий интернет-адрес после перехода по гиперссылке будет удален из истории веб-обозревателя, если `false` — текущий интернет-адрес сохранится в истории (по умолчанию: `false`);
- `state` — произвольное значение, подлежащее передаче целевому компоненту. Может быть любого типа.

> ***Внимание!***
>
> Данные из пропа `state` может передавать только маршрутизатор Browser Router (см. *разд. 11.2.4.1*). Маршрутизатор Hash Router эту возможность не поддерживает.

- любые произвольные пропы — они будут переданы формирующему гиперссылку тегу `<a>`:

```
<Link to="/items/add" className="navbar-item">Создать дело</Link>
```

Еще пару пропов компонента `Link` мы изучим позже.

Упрощенные гиперссылки обычно применяются в различных перечнях для перехода на экраны со сведениями о той или иной позиции такого перечня.

11.3.1.1.1. Написание путей в гиперссылках

Путь в пропе `to` или свойстве `pathname` простого объекта, указываемого в этом же пропе, может быть написан:

- с начальным слешем — отсчитывается от «корня» приложения:

```
// Модуль src\routes.js
<Route path="/" element={<App />}>
    <Route index element={<MainPage />} />
    <Route path="todos" element={<TodoList />} />
    <Route path="todos/add" element={<TodoAdd />} />
    <Route path="todos/undone" element={<UndoneTodoList />} />
</Route>

// Модуль src\TodoList.js
<Link to="/todos/add">Создать дело</Link>

// Модуль src\TodoAdd.js
<Link to="/">Главная страница</Link>
<Link to="/todos">Перечень дел</Link>
```

- без начального слеша — отсчитывается от текущего пути;

```
// Модуль src\TodoList.js
<Link to="add">Создать дело</Link>
```

Сегмент из двух точек (`..`), присутствующих в начале такого пути, обозначает, в зависимости от значения пропа `relative` компонента `Link`:

- `route` — «подъем» на предыдущий уровень иерархии маршрутов:

```
// Модуль src\TodoAdd.js
<Link to=".." relative="route">Главная страница</Link>
<Link to="../todos/undone" relative="route">
    Перечень невыполненных дел
</Link>
```

Поскольку `route` — значение пропа `relative` по умолчанию, указывать его необязательно:

```
<Link to="..">Главная страница</Link>
```

- `path` — «подъем» на один уровень иерархии путей (или, другими словами, удаление из пути одного сегмента справа):

```
// Модуль src\TodoAdd.js
<Link to=".." relative="path">Перечень дел</Link>
<Link to="../.." relative="path">Главная страница</Link>
<Link to="../undone" relative="path">
    Перечень невыполненных дел
</Link>
```

Если путь, указываемый в гиперссылке, должен содержать URL-параметры, для его формирования удобно применять функцию `generatePath()`:

```
generatePath([<шаблонный путь>, [<значения URL-параметров>={}]])
```

Шаблонный путь, на основе которого будет генерироваться реальный путь, должен быть указан в том виде, в котором он записан в соответствующем маршруте. *Значения URL-параметров* указываются в виде простого объекта, свойства которого соответствуют присутствующим в *шаблонном пути* URL-параметрам, а значения свойств задают значения этих параметров.

Функция возвращает полностью сформированный путь, который можно занести в гиперссылку.

Пример:

```
import { generatePath } from 'react-router-dom';
. . .
const path = generatePath('/todos/:key', { key: 1234 });
                                    // Результирующий путь: /todos/1234
. . .
<Link to={path}>Дело 1234</Link>
```

Если в указанном *шаблонном пути* присутствует URL-параметр `*` (звездочка — см. *разд. 11.2.1.1*), для его задания в объекте, содержащем значения URL-параметров, следует создать свойство с именем `'*'`:

```
const path2 = generatePath('/:userId/login/*',
      { userId: 'vasya', '*': 'and/redirect/to/todo/list' });
          // Результирующий путь: /vasya/login/and/redirect/to/todo/list
```

11.3.1.2. Полнофункциональные гиперссылки

Полнофункциональная гиперссылка, помимо всего прочего, способна выяснять, является ли она активной, и изменять при этом свое оформление указанным образом.

Полнофункциональную гиперссылку реализует компонент `NavLink`. Он поддерживает все пропы, характерные для компонента `Link` (см. *разд. 11.3.1.1* и *11.3.1.1.1*), а также приведенные далее:

- `className` — стилевые классы, указываемые у гиперссылки. Значение может быть указано в виде:
  - строки — как и у обычных HTML-тегов и других React-компонентов. Стилевые классы, как обычно, записываются через пробел;

- функции — принимающей в качестве единственного параметра простой объект со свойствами:
  - `isActive` — `true`, если текущая гиперссылка является активной, и `false` — в противном случае;
  - `isPending` — `true`, если в текущий момент производится навигация на другой экран, и `false` — в противном случае.

  Функция должна возвращать строку со стилевыми классами.

  Пример:

```
import { NavLink } from 'react-router-dom';
. . .
<NavLink to="/add"
    className={({ isActive }) =>
        'navbar-item' + (isActive ? ' is-active' : '')
    }
>
    Создать дело
</NavLink>
```

Значение по умолчанию: `undefined` (отсутствие любых стилевых классов);

- `style` — стиль, привязываемый к текущей гиперссылке. Может быть указан в виде:
  - простого объекта — как и у обычных HTML-тегов и других React-компонентов;
  - функции — принимающей в качестве единственного параметра простой объект со свойствами `isActive` и `isPending` (см. ранее). Функция должна возвращать стиль в виде простого объекта. Пример:

```
<NavLink to="/add"
    className={({ isActive }) =>
        ({
            color: (isActive ? 'blue' : 'white'),
            backgroundColor: (isActive ? 'white' : 'blue')
        })
    }
>
    Создать дело
</NavLink>
```

Значение по умолчанию: `undefined` (отсутствие стиля);

- `end` — если:
  - `false` — то в случае, когда текущий путь длиннее записанного в гиперссылке, гиперссылка считается активной, если записанный в ней путь совпадает с началом текущего пути;
  - `true` — в описанном случае гиперссылка *не* считается активной.

По умолчанию: `false`.

Примеры:

Гиперссылка	Совпадающие текущие пути
`<NavLink to="/todos"> . . . </NavLink>`	**/todos**, **/todos/1234**, **/todos/a/b/c/d**
`<NavLink to="/todos" end> . . . </NavLink>`	**/todos**

- `caseSensitive` — если `true`, при сравнении путей будет учитываться регистр символов, которыми они набраны, если `false` — *не* будет учитываться (по умолчанию: `false`).

Полнофункциональные гиперссылки обычно применяются в панелях навигации.

11.3.2. Перенаправление

Для перенаправления служит хук `useNavigate()`. Он вызывается без параметров.

Хук возвращает функцию, которая и будет выполнять перенаправление. Формат ее вызова таков:

<возвращенная функция>(*<путь>*[, *<настройки>*={}])

Путь, по которому следует произвести перенаправление, может быть задан в виде:

- строки:

```
import { useNavigate } from 'react-router-dom';
import { login } from './api.js';

export default function Login() {
    . . .
    const navigate = useNavigate();
    . . .
    const handleFormSubmit = async evt => {
        evt.preventDefault();
        const result = await login(email, password);
        if (!result)
            navigate('/');
    };
    . . .
}
```

Можно указывать пути как с начальным слешем, так и без такового (подробности — в *разд. 11.3.1.1.1*):

```
navigate('../todos/undone');
```

- простого объекта, описанного в разд. *11.3.1.1.1*;
- для перехода по истории веб-обозревателя — в виде целого числа, обозначающего положение требуемой позиции списка истории относительно текущей страницы.

Пример возврата на предыдущую страницу:

```
navigate(-1);
```

Пример перехода на две позиции списка истории вперед:

```
navigate(2);
```

В вызове *функции*, выполняющей перенаправление, можно указать *настройки*. Они задаются в виде простого объекта, свойства которого соответствуют отдельным настройкам. Доступны для указания следующие настройки:

- `relative` — аналогична одноименному пропу компонента `Link` (см. *разд. 11.3.1.1.1*). Принимается во внимание только при указании *пути*, содержащего сегменты из двух точек;
- `replace` — если `true`, текущий интернет-адрес после перенаправления будет удален из истории веб-обозревателя, если `false` — текущий интернет-адрес сохранится в истории (по умолчанию: `false`).

  Этой настройке следует давать значение `true` при перенаправлении с экранов, к которым у текущего посетителя не должно быть доступа (например, при перенаправлении гостя с экрана перечня дел на экран входа). В таком случае посетитель при нажатии кнопки **Назад** попадет не на недоступный ему экран, а на экран, на котором он был перед попыткой попасть на него. Пример:

```
if (!getCurrentUser())
    navigate('/login', { replace: true });
```

- `state` — произвольные данные, подлежащие передаче целевому компоненту.

11.4. Получение сведений о текущем интернет-адресе

11.4.1. Получение значений URL-параметров

Получить в коде компонента значения URL-параметров, присутствующих в текущем пути, можно с помощью хука `useParams()`. Он вызывается без параметров и возвращает простой объект, свойства которого соответствуют URL-параметрам. Пример:

```
import { useParams } from 'react-router-dom';
export default function TodoDetail() {
    const { key } = useParams();
    . . .
}
```

Если в текущем пути присутствует URL-параметр `*` (звездочка — см. *разд. 11.2.1.1*), в возвращенном объекте будет присутствовать свойство с именем `'*'`, хранящее значение этого URL-параметра:

```
const params = useParams();
const remaining = params['*'];
```

11.4.2. Получение и изменение значений GET-параметров

Для получения и изменения значений GET-параметров, присутствующих в текущем интернет-адресе, предназначен хук `useSearchParams()`:

```
useSearchParams(<изначальный набор GET-параметров>)
```

Изначальный набор GET-параметров можно задать в виде:

- строки:

```
import { useSearchParams } from 'react-router-dom;
. . .
const res = useSearchParams('keyword=react&count=10');
```

- простого объекта, свойства которого представляют отдельные GET-параметры:

```
const res2 = useSearchParams({ keyword: 'react', 'count': 10);
```

- объекта класса `URLSearchParams`, встроенного в JavaScript.

GET-параметры, заданные в хуке, будут добавлены к текущему интернет-адресу и появятся в строке адреса веб-обозревателя.

Хук `useSearchParams()` возвращает массив из двух элементов:

- набора текущих GET-параметров, представленного объектом класса `URLSearchParams`:

```
const [searchParams, setSearchParams] =
                    useSearchParams('keyword=react&count=10');
const keyword = searchParams.get('keyword');
const cnt = searchParams.get('count');
```

  В процессе выполнения функции компонента значения GET-параметра из этого набора не изменяются;

- *сеттера GET-параметров* — функции, при вызове изменяющей текущий набор GET-параметров.

  Сеттер GET-параметров в качестве единственного параметра может принимать:

  - новый набор GET-параметров в одном из форматов, приведенных ранее:

```
setSearchParams({ keyword: 'JavaScript', 'count': cnt });
```

  - функцию, принимающую с единственным параметром текущий набор GET-параметров в виде объекта класса `URLSearchParams` и возвращающую новый набор в одном из форматов, приведенных ранее:

```
setSearchParams(currentSP => {
    return {
        keyword: 'React 19',
        'count': currentSP.get('count') + 1
    }
});
```

Результата сеттер GET-параметров возвращать не должен.

Изменения, сделанные в GET-параметрах вызовом сеттера, тут же отразятся на интернет-адресе, выводящемся в панели адреса веб-обозревателя.

11.4.3. Получение текущего местоположения

Местоположение в терминологии React Router — это совокупность сведений о текущем интернет-адресе и значения, переданного целевому компоненту через проп `state` компонентов `Link` (см. *разд. 11.3.1.1*) и `NavLink` (см. *разд. 11.3.1.2*) или настройку `state` перенаправления (см. *разд. 11.3.2*).

Получить местоположение можно вызовом хука `useLocation()` без параметров.

Хук возвращает простой объект со следующими свойствами:

- `pathname` — путь, присутствующий в текущем интернет-адресе, в виде строки;
- `search` — набор GET-параметров с начальным символом `?` в виде строки;
- `hash` — якорь с начальным символом решетки `#` в виде строки;
- `state` — значение, переданное текущему компоненту через проп `state` гиперссылки или одноименную настройку перенаправления:

```
// Модуль src\routes.js
<Route path='add' element={<TodoAdd />} state="Добавить дело" />

// Модуль src\TodoAdd.js
import { useLocation } from 'react-router-dom';
. . .
const { state } = useLocation();
```

- `key` — уникальный идентификатор состояния, сгенерированный самим React Router, в виде строки. Может использоваться, например, для сохранения в хранилище Web API данных, относящихся к текущему местоположению.

Объект местоположения создается хуком `useLocation()` при монтировании компонента и в дальнейшем не изменяется.

При навигации на другой экран щелчком на гиперссылке или при перенаправлении всегда создается новый объект местоположения с новым идентификатором — даже если навигация была произведена на ранее посещенный экран. Новый объект местоположения сохраняется в истории веб-обозревателя.

Напротив, при навигации нажатием кнопки **Назад** или **Вперед** всегда используется существующий объект местоположения, извлеченный из истории.

11.5. Дополнительные программные инструменты React Router

11.5.1. Разметочные маршруты

Разметочный маршрут — это узловой маршрут (см. *разд. 11.2.2.2.2*), не содержащий фрагмента пути. В разметочном маршруте проп `path` вообще не указывается.

Поскольку разметочный маршрут не содержит фрагмента пути, он ничего не добавляет в результирующие пути находящихся в нем дочерних маршрутов.

Разметочные маршруты применяются для объединения части маршрутов с какой-либо целью. Такой целью может быть, например, вывод целевых компонентов, указанных в дочерних маршрутах, в заданном разметочном компоненте. Вот пример:

```
const rm = (
    <Route path="/" element={<App />}>
        . . .
        <Route element={<UserLayout />}>
            <Route path="login" element={<Login />} />
            <Route path="logout" element={<Logout />} />
            <Route path="register" element={<Register />} />
        </Route>
    </Route>
);
```

Здесь показан разметочный маршрут с компонентом `UserLayout`, в который вложены три маршрута. Целевые компоненты, указанные в этих маршрутах, будут выводиться внутри компонента `UserLayout`, записанного в разметочном маршруте.

При этом в пути дочерних маршрутов никаких фрагментов добавлено не будет. На компонент `Login` будет вести путь **/login**, на компонент `Logout` — путь **/logout**, а на компонент `Register` — путь **/register**.

11.5.2. Обработка события *beforeunload* окна

Иногда бывает необходимо обработать событие `beforeunload` окна веб-обозревателя, возникающего перед переходом с текущей страницы на другую. В обработчике этого события можно, например, сохранить занесенные посетителем данные перед переходом на другую страницу, чтобы восстановить их после возврата.

Обработать событие `beforeunload` можно посредством хука `useBeforeUnload()`:

```
useBeforeUnload(<обработчик события>)
```

Задаваемый *обработчик события* не должен ни принимать параметров, ни возвращать результата.

Обработчик привязывается к событию при монтировании компонента и убирается перед его размонтированием.

Для повышения производительности задаваемый обработчик желательно мемоизировать с применением хука `useCallback()` (см. *разд. 7.1.2.1*).

Если передать хуку `useBeforeUnload()` другой обработчик, предыдущий обработчик будет убран, а новый — привязан к событию.

Вот пример сохранения заголовка и описания дела, введенных посетителем, перед уходом на другую страницу и восстановления их по возвращении:

```
import { useBeforeUnload } from 'react-router-dom';

export default function TodoAdd() {
    . . .
    useBeforeUnload(
        useCallback(() => {
            window.localStorage.setItem('title', title);
            window.localStorage.setItem('desc', desc);
        }, [title, desc]);
    );

    useEffect(() => {
        if (!title && window.localStorage.getItem('title'))
            setTitle(window.localStorage.getItem('title'));
        if (!desc && window.localStorage.getItem('desc'))
            setDesc(window.localStorage.getItem('desc'));
    });
}
```

11.5.3. Блокировка ухода

Может возникнуть необходимость выполнить *блокировку ухода* — предотвратить уход посетителя с текущего экрана на какой-либо другой экран или даже другую страницу. Обычно такая блокировка накладывается при истинности заданного условия, и посетителю предоставляется возможность все же покинуть экран[1].

Реализовать блокировку ухода позволяет хук `useBlocker()`:

```
useBlocker(<блокировать уход?>|<функция, управляющая блокировкой>)
```

Если параметр *блокировать уход* имеет значение `true`, блокировщик всегда будет блокировать уход, если `false` — не будет блокировать никогда.

Функция, управляющая блокировкой, должна принимать в качестве параметра простой объект со следующими свойствами:

- `currentLocation` — текущее местоположение;
- `nextLocation` — местоположение, по которому посетитель пытается перейти.

  Значения обоих этих свойств представляются в виде простых объектов, описанных в *разд. 11.4.3*;

[1] Предполагается, что посетитель знает, что делает.

- `historyAction` — характер действия, предпринятого посетителем для ухода со страницы, в виде строкового обозначения:
  - `'PUSH'` — щелчок на гиперссылке или какое-либо действие, вызвавшее перенаправление, с сохранением текущего местоположения в истории веб-обозревателя;
  - `'REPLACE'` — то же самое, что и `'PUSH'`, только без сохранения текущего местоположения в истории;
  - `'POP'` — перемещение на предыдущую или следующую позицию в истории веб-обозревателя нажатием кнопки **Назад** или **Вперед**.

Возвращать *функция* должна значение `true` или `false`, тем самым накладывая или отменяя блокировку ухода. *Функция, управляющая блокировкой*, будет вызываться перед попыткой посетителя уйти с текущего экрана.

Хук `useBlocker()` возвращает простой объект со следующими свойствами:

- `state` — состояние блокировки в виде одной из строк:
  - `'unblocked'` — блокировка ухода *не* наложена;
  - `'blocked'` — блокировка ухода наложена, и блокировщик ждет решения посетителя, оставаться ли на текущем экране или уйти с него;
  - `'proceeding'` — посетитель принял решение уйти с текущего экрана, и блокировщик произвел уход;
- `location` — в зависимости от состояния:
  - `'blocked'` и `'proceeding'` — местоположение, по которому посетитель хочет перейти;
  - `'unblocked'` — значение `undefined`.

Еще возвращенный объект содержит два метода, существующие только в состоянии `'blocked'` и не возвращающие результата:

- `proceed()` — предписывает блокировщику все же выполнить уход с экрана;
- `reset()` — предписывает блокировщику остаться на текущем экране.

Вот пример блокировки ухода в случае, если в веб-форму занесены данные, с выводом непосредственно на странице предупреждения с кнопками **ОК** и **Отмена**:

```
import { useBlocker } from 'react-router-dom';

export default function TodoAdd() {
    . . .
    const blocker = useBlocker(
        ({ currentLocation, nextLocation }) =>
            (title || desc) &&
                currentLocation.pathname != nextLocation.pathname
    );
    . . .
```

```
    return (
        <section>
            . . .
            {blocker.state == 'blocked' && (
                <div>
                    <input type="button" value="Остаться'
                           onClick={() => {blocker.reset()}} />
                    <input type="button" value="Перейти"
                           onClick={() => {blocker.proceed()}} />
                </div>
            )}
        </section>
    );
}
```

11.5.4. Проверка пути

Есть возможность проверить, совпадает ли заданный *путь* с указанным *шаблонным путем*. Это выполняется с помощью функции `matchPath()`:

`matchPath(<`*шаблонный путь*`>, <`*проверяемый путь*`>)`

Шаблонный путь может быть задан в виде:

- строки;
- простого объекта со свойствами:
  - `path` — собственно шаблонный путь в виде строки;
  - `caseSensitive` — если `true`, при проверке путей будет учитываться регистр символов, которыми они набраны, если `false` — не будет учитываться (по умолчанию: `false`);
  - `end` — если:
    - `false` — то в случае, когда *проверяемый путь* длиннее *шаблонного*, *пути* считаются совпадающими, если *шаблонный путь* совпадает с началом *проверяемого пути*;
    - `true` — в описанном случае *пути* считаются несовпадающими.

    По умолчанию: `false`.

Проверяемый путь задается в виде строки.

Функция `matchPath()` возвращает:

- в случае совпадения *путей* — простой объект со следующими свойствами:
  - `pathname` — совпавшая часть *проверяемого пути* в виде строки;
  - `params` — простой объект с URL-параметрами, аналогичный возвращаемому хуком `useParams()` (см. *разд. 11.4.1*);

- `pattern` — простой объект, аналогичный задаваемому в вызове функции `matchPath()` первым параметром. Если в первом параметре этой функции передан:
  - строковый путь — в свойстве `path` возвращенного объекта будет содержаться заданный путь, а свойства `caseSensitive` и `end` получат значения по умолчанию;
  - простой объект — возвращенный объект будет его копией;

♦ в случае несовпадения *путей* — значение `null`.

Пример:

```
import { matchPath } from 'react-router-dom';
. . .
const match = matchPath('/items/add', window.location.pathname);
. . .
const match2 = matchPath({ path: '/items/add', end: false },
                         window.location.pathname);
```

Также имеется возможность проверить, совпадает ли *текущий* путь с указанным *шаблонным путем*, использовав хук `useMatch()`:

```
useMatch(<шаблонный путь>)
```

Шаблонный путь задается в том же формате, что и в вызове функции `matchPath()`, и точно такой же возвращается результат. Пример:

```
import { useMatch } from 'react-router-dom';
. . .
const match3 = useMatchPath('/items/:key');
if (match3) {
    const key = match3.params.key;
    . . .
}
```

11.5.5. Задание маршрутов в виде простых объектов

В *разд. 11.2.1* и *11.2.2.1* рассматривалось написание списка маршрутов в виде структуры из компонентов `Route`. Однако список маршрутов можно описать структурой, составленной из простых объектов JavaScript. Такой список значительно менее нагляден, однако несколько быстрее обрабатывается, поскольку отпадает необходимость в его предварительном преобразовании вызовом функции `createRoutesFromElements()`.

Список маршрутов, представленных простыми объектами, должен быть организован в виде массива. Отдельные простые объекты-маршруты должны содержать свойства, соответствующие пропам компонента `Route` (см. *разд. 11.2.1*).

Простые объекты, представляющие дочерние маршруты, следует объединить в массив и присвоить свойству `children` объекта — родительского маршрута.

Пример простого списка маршрутов, записанного в виде массива простых объектов:

```
import { createBrowserRouter } from 'react-router-dom';

const rs = [
    { path: '/', element: <TodoList /> },
    { path: '/add', element: <TodoAdd /> },
    { path: '/:key', element: <TodoDetail /> },
    { path: '/login', element: <Login /> },
    { path: '/logout', element: <Logout /> },
    { path: '/register', element: <Register /> }
];

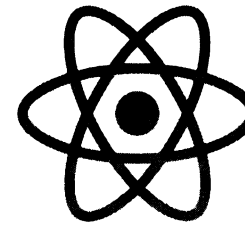
const router = createBrowserRouter(rs);
```

Пример иерархического списка маршрутов, записанного аналогичным образом:

```
const rs2 = [
    { path: '/', element: <App />, children: [
        { path: '', element: <TodoList /> },
        { path: 'add', element: <TodoAdd /> },
        { path: ':key', element: <TodoDetail /> },
        { path: 'user', element: <UserLayout />, children: [
            { index: true, element: <Profile /> },
            { path: 'login', element: <Login /> },
            { path: 'logout', element: <Logout /> },
            { path: 'register', element: <Register /> }
        ] }
    ] }
];
```

Какой способ применить для написания списка маршрутов — вопрос вкуса. Автор предпочитает способ с использованием компонентов `Route`, полагая его более наглядным.

ГЛАВА 12



Библиотека React Router: навигация, раздел 2

12.1. Загрузка и отправка данных

Маршрутизатор React Router имитирует навигацию по обычным, многостраничным сайтам. При попытке перейти на другой экран приложения он создает клиентский запрос (в виде объекта класса `Request`, встроенного в JavaScript), сам обрабатывает его и выдает серверный ответ (объект встроенного класса `Response`), который и вызывает переход на другой экран.

Кроме того, маршрутизатор способен при получении клиентского запроса по заданному маршруту выполнить указанный программный код. Этот код может как загружать данные, требуемые целевому компоненту, из какого-либо внешнего источника (бэкенда, базы данных IndexedDB, хранилища Web API, обычного массива JavaScript и т. п.), так и записывать в него данные, переданные текущим компонентом.

12.1.1. Загрузка данных. Загрузчики

Загрузчик — программный код, выполняющийся при получении по заданному маршруту клиентского запроса, сформированного с применением HTTP-метода GET. Такой запрос формируется при переходе по гиперссылке или программном перенаправлении.

Загрузчик практически всегда применяется для загрузки данных из внешнего хранилища с целью передачи их целевому компоненту.

12.1.1.1. Написание загрузчиков

Загрузчик реализуется в виде обычной функции JavaScript. Функция может быть асинхронной.

Функция должна принимать единственным параметром простой объект со свойствами:

- `request` — клиентский запрос, получение которого вызвало запуск загрузчика, в виде объекта встроенного класса `Request`;

- `params` — простой объект с URL-параметрами, аналогичный выдаваемому хуком `useParams()` (см. *разд. 11.4.1*).

В качестве результата загрузчик может вернуть:

- произвольное значение:

```
export async function getTodo({ params }) {
    const rawTodo = await loadTodo(currentUserId, params.key);
    const data = { title: rawTodo.title, desc: rawTodo.desc };
    return data;
}
```

- полноценный серверный ответ в виде объекта встроенного класса `Response`:

```
export async function getTodo({ params }) {
    . . .
    const response = new Response(
        JSON.stringify(data),
        {
            status: 200,
            headers: {
                'Content-Type': 'application/json; utf-8'
            }
        }
    );
    return response;
}
```

Упростить код, создающий серверный ответ с данными в формате JSON, позволит функция `json()`:

```
json(<передаваемое значение>[, <настройки>={}])
```

Передаваемое значение указывается как есть, в незакодированном виде. *Настройки* задаются в виде простого объекта, как и у конструктора класса `Response`.

Пример:

```
import { json } from 'react-router-dom';
export async function getTodo({ params }) {
    . . .
    return json(data, { status: 200 });
}
```

Как правило, код загрузчиков (а также описанных в *разд. 12.1.2* воздействий) сохраняется в отдельном модуле (который можно назвать, например, src\api.js). Все загрузчики (и воздействия) подвергаются множественному экспорту.

12.1.1.1.1. Перенаправление в загрузчиках

Перенаправление по другому пути в загрузчике можно реализовать, вернув в качестве результата «пустой» серверный ответ с кодом статуса 302 и заголовком

Location, содержащим требуемый путь. Вот пример перенаправления на экран входа в случае, если посетитель еще не вошел в приложение:

```
if (!currentUserId)
    return new Response('', {
        status: 302,
        headers: { Location: '/login' }
    });
```

Функция redirect() позволяет значительно упростить код, выполняющий перенаправление:

```
redirect(<путь>[, <настройки>={}])
```

Путь, по которому необходимо выполнить перенаправление, указывается в виде строки. *Настройки* задаются в виде простого объекта, как и у конструктора класса Response.

Пример:

```
import { redirect } from 'react-router-dom';
. . .
if (!currentUserId)
    return redirect('/login');
```

12.1.1.2. Указание загрузчика в маршруте

Загрузчик указывается в маршруте с нужным шаблонным путем в пропе loader компонента Route. Пример:

```
import { getTodo } from './api.js';
. . .
<Route path=":key" element={<TodoDetail />} loader={getTodo} />
```

Теперь после перехода по пути формата */<идентификатор дела>* загрузчик getTodo() загрузит дело с полученным в составе пути *идентификатором* и передаст его целевому компоненту TodoDetail для вывода.

12.1.1.3. Получение результата, возвращенного загрузчиком

Получить в коде целевого компонента результат, возвращенный загрузчиком, который записан в маршруте, указывающем на этот компонент, можно, вызвав хук useLoaderData() без параметров:

```
import { useLoaderData } from 'react-router-dom';
export default function TodoDetail() {
    const todo = useLoaderData();
    . . .
}
```

Также есть возможность получить результат, возвращенный загрузчиком, записанным в любом из родительских маршрутов. Для этого необходимо выполнить два действия:

1. Присвоить нужному родительскому маршруту якорь, воспользовавшись пропом `id`. Якорь может быть произвольным, однако он должен быть уникальным в пределах списка маршрутов.
2. Использовать для получения результата, возвращенного загрузчиком, указанным в маршруте с заданным *якорем*, хук `useRouteLoaderData(<якорь>)`.

Пример:

```
// Модуль src\router.js
import { getCurrentUser, getTodo } from './api.js';
. . .
<Route path="/" element={<App />} loader={getCurrentUser} id="root">
    <Route path=":key" element={<TodoDetail />} loader={getTodo} />
    . . .
</Route>

// Модуль src\TodoDetail.js
import { useLoaderData, useRouteLoaderData } from 'react-router-dom';
export default function TodoDetail() {
    const todo = useLoaderData();
    const currentUser = useRouteLoaderData('root');
    . . .
}
```

12.1.2. Отправка данных. Воздействия React Router

Воздействие React Router[1] — это фрагмент кода, выполняющийся при получении по заданному маршруту клиентского запроса HTTP-методом POST, PUT, PATCH или DELETE. Такой запрос формируется при отправке данных из веб-формы, вызванной нажатием кнопки отправки данных или программно.

Воздействие практически всегда применяется для занесения во внешнее хранилище данных, отправленных компонентом, с которого выполняется переход (эти данные могут как заноситься посетителем, так и вырабатываться компонентом самостоятельно).

12.1.2.1. Написание воздействий React Router

Воздействие реализуется в виде обычной функции JavaScript. Функция может быть асинхронной.

Функция должна принимать такой же параметр и возвращать такой же результат, как и загрузчик (см. *разд. 12.1.1.1*).

Объект запроса, доступный из свойства `request` полученного с параметром простого объекта, хранит данные, переданные воздействию. Формат этих данных зависит от настроек, указанных при их передаче. По умолчанию передаваемые воздействию

[1] Не путайте с аналогичными им воздействиями React, описанными в *разд. 8.2.2*.

данные представляются в кодировке `application/x-www-form-urlencoded` и хранятся в объекте класса `FormData`, встроенного в JavaScript.

Вот пример воздействия React Router, создающего новое дело, производящего перенаправление в случае успеха и возвращающего код ошибки, если возникла нештатная ситуация:

```
export async function addTodo({ request, params }) {
    const fd = await request.formData();
    const error = await saveTodo(fd);
    return (error) ? error : redirect('/');
}
```

12.1.2.2. Указание воздействия React Router в маршруте

Воздействие указывается в маршруте с нужным шаблонным путем в пропе `action` компонента `Route`. Пример:

```
import { addTodo } from './api.js';
. . .
<Route path="add" element={<TodoAdd />} action={addTodo} />
```

Теперь после перехода по пути **/add** воздействие `addTodo()` получит сведения о создаваемом деле, отправленные ей компонентом, создаст новое дело и произведет перенаправление на перечень дел (или вернет код ошибки, если что-то пошло не так).

12.1.2.3. Инициирование запуска воздействия React Router

Чтобы запустить воздействие с передачей ему данных, подлежащих обработке, следует отправить клиентский запрос по пути, указанном в маршруте, где оно записано. Клиентский запрос следует выполнить HTTP-методом POST, PUT, PATCH или DELETE (при использовании HTTP-метода GET будет запущен заданный в маршруте загрузчик).

Поддерживаются два способа запустить воздействие, записанное в маршруте. Не забываем, что при этом одновременно будет произведена навигация на указанный в этом маршруте компонент.

12.1.2.3.1. Запуск воздействия из веб-формы

Самый простой способ произвести запуск воздействия — указать в веб-форме необходимый путь и некоторые другие параметры отправляемого клиентского запроса.

Для создания веб-формы, запускающей воздействие, следует использовать не HTML-тег `<form>`, а компонент `Form`[1], предоставляемый React Router. Этот компонент поддерживает следующие пропы:

[1] Будьте внимательны: имя компонента начинается с большой буквы, а имя тега — нет.

- `action` — путь, по которому следует отправить клиентский запрос, в виде строки. Может быть с начальным слешем или без такового (подробности — в *разд. 11.3.1.1.1*). По умолчанию: текущий путь;
- `relative` — аналогичен одноименному пропу компонентов гиперссылок (см. *разд. 11.3.1.1.1*). Принимается во внимание только при указании пути без начального слеша;
- `method` — название HTTP-метода, с помощью которого создается отправляемый клиентский запрос, в виде строки (по умолчанию: `'get'`).

> ***Внимание!***
>
> Не забывайте указывать HTTP-метод в пропе `method` компонента `Form`. Если вы его не укажете, для отправки запроса будет использован HTTP-метод GET, и воздействие не запустится (а запустится загрузчик, если он задан в маршруте).

- `replace` — если `true`, текущий интернет-адрес после навигации на целевой компонент будет удален из истории веб-обозревателя, если `false` — текущий интернет-адрес сохранится в истории (по умолчанию: `false`);
- `state` — произвольное значение любого типа, подлежащее передаче целевому компоненту. Может быть получено в целевом компоненте вызовом хука `useLocation()` (см. *разд. 11.4.3*).

Еще пару пропов мы изучим позже.

Кроме того, у элементов управления, данные из которых должны быть отправлены воздействию, обязательно следует указать наименования — в атрибутах тега `name`. Если этого не сделать, данные отправлены не будут.

При нажатии кнопки отправки данных в такой веб-форме данные, занесенные в находящиеся в ней элементы управления с указанными наименованиями, заносятся во вновь созданный объект класса `FormData` и отправляются воздействию. Все это выполняется автоматически — нам никаких действий для этого производить не придется. Если не предполагается реализовывать валидацию данных, нет необходимости даже обрабатывать элементы управления в стиле React (см. *разд. 8.1*).

Вот пример компонента с веб-формой, отправляющей данные воздействию `addTodo()`, написанному ранее:

```
import { Form } from 'react-router-dom';
export default function TodoAdd() {
    return (
        <Form action="/add" method="post">
            <label>Заголовок</label>
            <input name="title" />
            <label>Примечание</label>
            <textarea name="desc" />
            <input type="submit" value="Создать дело" />
        </Form>
    );
}
```

Преимущество этого способа запуска воздействия — простота. Но, к сожалению, нет возможности произвести какую бы то ни было дополнительную обработку занесенных в веб-форму данных перед их отправкой. Если же отправляемые данные необходимо предварительно обработать, следует использовать второй способ — программный запуск воздействия.

12.1.2.3.2. Программный запуск воздействия

Этот способ позволяет произвести предварительную обработку данных перед отправкой воздействию и даже отослать ему произвольные данные. При его использовании не нужно применять компонент `Form` — можно обойтись обычной веб-формой, создаваемой HTML-тегом `<form>`.

Программный запуск воздействия выполняется вызовом особой функции. Получить ее можно вызовом хука `useSubmit()` без параметров.

Возвращенная хуком *функция* вызывается в следующем формате:

<функция>(*<отправляемое значение>*[, *<настройки>*={}])

В качестве *отправляемого значения* можно указать:

- значение произвольного типа — оно будет преобразовано в указанный в *настройках* формат (по умолчанию: `application/x-www-form-urlencoded`):

```
import { useSubmit } from 'react-router-dom';

export default function TodoAdd() {
    . . .
    const submit = useSubmit();

    const handleFormSubmit = evt => {
        evt.preventDefault();
        submit({ title, desc, image },
               { action: '/add', method: 'post' });
    };

    return (
        <form onSubmit={handleFormSubmit}>
            . . .
        </form>
    );
}
```

- объект класса `FormData` — будет передан воздействию без всяких преобразований:

```
const handleFormSubmit = evt => {
    evt.preventDefault();
    const fd = new FormData();
    fd.append('title', title);
    fd.append('desc', desc);
```

```
    fd.append('image', image);
    submit(fd, { action: '/add', method: 'post' });
};
```

- объект класса `URLSearchParams` — будет преобразован в объект класса `FormData`:

```
const fd = new URLSearchParams();
fd.append('title', title);
fd.append('desc', desc);
fd.append('image', image);
submit(fd, { action: '/add', method: 'post' });
```

- данные в любом формате, поддерживаемом конструктором класса `URLSearchParams`, — будут преобразованы в объект класса `FormData`:

```
const fd = [['title', title], ['desc', desc], ['image', image]];
submit(fd, { action: '/add', method: 'post' });
```

Настройки указываются в виде простого объекта, свойства которого соответствуют отдельным настройкам. Поддерживаются следующие настройки:

- `encType` — обозначение кодировки отправляемых данных в виде строки. Поддерживаются следующие кодировки:
  - `'application/x-www-form-urlencoded'` — для извлечения отправленных данных в коде воздействия следует использовать асинхронный метод `formData()` класса `Response`:
  - `'application/json'` — асинхронный метод `json()`:

```
// Модуль src\TodoAdd.js
export default function TodoAdd() {
    . . .
    const handleFormSubmit = evt => {
        evt.preventDefault();
        submit({ title, desc, image },
               { action: '/add', method: 'post',
                 endType: 'application/json' });
    };
    . . .
}

// Модуль src\api.js
export async function addTodo({ request, params }) {
    const fd = await request.json();
    . . .
}
```

  - `'text/plain'` — асинхронный метод `text()`;
- `action`, `relative`, `method`, `replace` и `state` — соответствуют одноименным пропам компонента `Form` (см. *разд. 12.1.2.3.1*).

12.1.2.4. Получение результата, возвращенного воздействием React Router

Получить результат, выданный выполнившимся воздействием, в целевом компоненте можно, вызвав хук `useActionData()` без параметров. Пример:

```
import { useActionData, useSubmit } from 'react-router-dom';

export default function TodoAdd() {
    . . .
    const error = useActionData();
    const submit = useSubmit();

    const handleFormSubmit = evt => {
        evt.preventDefault();
        submit({ title, desc, image },
               { action: '/add', method: 'post' });
    };

    return (
        <form onSubmit={handleFormSubmit}>
            . . .
        </form>
        {error && <p>{error}</p>}
    );
}
```

12.1.3. Фетчеры

Фетчер — это программный механизм, позволяющий запускать загрузчики и воздействия путем отправки клиентских запросов *без навигации на другой экран*.

12.1.3.1. Создание фетчеров

Фетчер создается вызовом хука `useFetcher()`:

```
useFetcher([<настройки>={}])
```

Хук возвращает объект, представляющий созданный фетчер.

Настройки указываются в виде простого объекта, свойства которого соответствуют отдельным настройкам. В настоящее время поддерживается только настройка `key`, задающая уникальный строковый ключ, идентифицирующий создаваемый фетчер. Зачем нужен этот ключ, мы узнаем чуть позже.

Объект фетчера — это реактивные данные. Изменение состояния фетчера (например, получение данных от загрузчика) вызывает обновление компонента.

Пример:

```
import { useFetcher } from 'react-router-dom';
export default function TodoList() {
    const fetcher = useFetcher();
    . . .
}
```

Фетчер создается при монтировании компонента, не изменяется при его обновлениях и удаляется при размонтировании.

12.1.3.2. Загрузка данных с помощью фетчеров

При загрузке данных фетчер отправляет клиентский запрос HTTP-методом GET по заданному пути. В результате загрузчик, который записан в маршруте, совпавшем с этим путем, запускается, загружает и возвращает данные, которые фетчер благополучно получает.

Загрузка данных инициируется вызовом у объекта фетчера, полученного ранее, метода `load()`:

```
load(<путь>)
```

Путь, по которому находится загрузчик, предоставляющий требуемые данные, указывается в виде строки.

Метод `load()` не возвращает результата.

Для получения запрошенных данных служат следующие свойства объекта фетчера:

- `data` — собственно результат, выданный загрузчиком;
- `state` — обозначение состояния фетчера в виде строки: `'idle'` (фетчер простаивает) или `'loading'` (фетчер отправил запрос загрузчику и теперь ждет от него результата).

Пример:

```
import { useEffect } from 'react';
import { useFetcher } from 'react-router-dom';

export default function TodoList() {
    const fetcher = useFetcher();

    useEffect(() => {
        if (fetcher.state == 'idle' && !fetcher.data)
            fetcher.load('/api/todos');
    }, [fetcher]);

    return (
        <table>
            {fetcher.data && fetcher.data.map(item => (
                <tr key={item.key}>
                    . . .
```

```
                </tr>
            ))}
        </table>
    );
}
```

12.1.3.3. Отправка данных с помощью фетчера

При отправке данных фетчер посылает клиентский запрос HTTP-методом POST, PUT, PATCH или DELETE по заданному пути. Воздействие из маршрута, совпавшего с этим путем, запускается, получает, обрабатывает и сохраняет данные, после чего, возможно, выдает фетчеру результат.

Выполнить отправку данных с помощью фетчера можно тремя способами.

12.1.3.3.1. Отправка данных посредством веб-формы фетчера

Проще всего отправить данные, воспользовавшись особой веб-формой, автоматически создаваемой объектом фетчера.

Компонент веб-формы фетчера хранится в свойстве `Form` объекта фетчера. Этот компонент полностью аналогичен компоненту `Form` (см. *разд. 12.1.2.3.1*), за тем исключением, что не выполняет навигацию.

Пример:

```
import { useFetcher } from 'react-router-dom';
export default function Login() {
    . . .
    const fetcher = useFetcher();
    . . .
    return (
        <fetcher.Form action="/login" method="post">
            . . .
        </fetcher.Form>
    );
}
```

Получить результат обработки данных, выданный воздействием, можно с помощью следующих свойств объекта фетчера:

- `data` — собственно результат, выданный воздействием;
- `state` — обозначение состояния фетчера в виде строки: `'idle'` (фетчер простаивает) или `'submitting'` (фетчер отправил запрос с данными воздействию и теперь ждет от него результата);
- `formData` — данные из веб-формы, представленные объектом класса `FormData`;
- `formAction` — путь, указанный в пропе `action` компонента веб-формы, в виде строки;
- `formMethod` — обозначение метода кодирования данных, указанное в пропе `method` компонента веб-формы, в виде строки.

Пример:

```
import { useFetcher, useNavigate } from 'react-router-dom';

export default function Login() {
    . . .
    const navigate = useNavigate()
    const fetcher = useFetcher();

    const error = fetcher.data;
    if (error == 'none')
        navigate('/');
    . . .
    return (
        <div>
            <fetcher.Form action="/login" method="post">
                . . .
            </fetcher.Form>
            {error && <p>{fetcher.data}</p>}
        </div>
    );
}
```

12.1.3.3.2. Упрощенный способ отправки данных фетчером из веб-формы

Если не требуется получать результат от воздействия, можно применить упрощенный способ отправки данных из веб-формы с использованием фетчера. Для этого следует создать веб-форму на основе компонента `Form` (см. *разд. 12.1.2.3.1*) и указать в ней проп `navigate` со значением `false`.

Пример:

```
import { Form } from 'react-router-dom';
. . .
<Form . . . navigate={false}>
    . . .
</Form>
```

При нажатии кнопки отправки данных в такой веб-форме React Router неявно создаст фетчер и использует его для отправки данных.

В пропе `fetcherKey` компонента `Form` можно указать уникальный ключ для создаваемого библиотекой фетчера.

12.1.3.3.3. Программная отправка данных фетчером

Посредством фетчера можно отправить и произвольные данные. Для этого служит метод `submit()` объекта фетчера, имеющий тот же формат вызова, что и функция, возвращенная хуком `useSubmit()` (см. *разд. 12.1.2.3.2*).

Пример:

```
import { useFetcher } from 'react-router-dom';

export default function Register() {
    . . .
    const fetcher = useFetcher();

    const handleFormSubmit = evt => {
        evt.preventDefault();
        fetcher.submit({email, password},
                       {action: '/register', method: 'post'});
    };
    . . .
}
```

12.1.3.4. Доступ к произвольному фетчеру

В любом компоненте можно получить доступ к любому из фетчеров, созданных во всех имеющихся в приложении компонентах.

Прежде всего, при создании фетчера хуком `useFetcher()` следует указать ключ — в настройке `key` (см. *разд. 12.1.3.1*). Ключ можно выбрать произвольно, однако не следует забывать, что он должен быть уникальным в пределах приложения.

Чтобы в другом компоненте получить доступ к фетчеру, зная его ключ, достаточно вызвать хук `useFetcher()`, также задав ключ в настройке `key`.

Пример:

```
// Модуль src\TodoSetDone.js
import { useFetcher } from 'react-router-dom';

function TodoSetDone({ item }) {
    const fetcher = useFetcher({ key: 'set-done-fetcher' });

    const buttonClickHandler = () => {
        fetcher.submit(null, { action: `/${item.key}`,method: 'patch' });
    };

    return (
        <input type="button" value="Дело сделано!"
                             onClick={buttonClickHandler} />
    );
}

// Модуль src\TodoList.js
export default function TodoList() {
    const fetcher = useFetcher({ key: 'set-done-fetcher' });
    const fetchSt = fetcher.state;
    . . .
```

```
    return (
        <section style={{ opacity: fetchSt == 'submitting' ? .2 : 1 }}>
            {list.map(item => (
                . . .
                <TodoSetDone item={item} />
                . . .
            ))}
        </section>
    );
}
```

Если при создании фетчера не указать ключ, React Router сгенерирует его самостоятельно. Вот только извлечь этот ключ, чтобы получить доступ к фетчеру из другого компонента, мы не сможем.

12.1.3.5. Получение сведений обо всех созданных в приложении фетчерах

Получить сведения обо всех созданных в приложении фетчерах можно вызовом хука `useFetchers()` без параметров. Хук возвращает массив, который содержит объекты, аналогичные объектам фетчеров (см. *разд. 12.1.3.2* и *12.1.3.3*), но не включающие свойство `Form`, методы `load()` и `submit()`.

Вот пример получения массива созданных фетчеров и поиска в нем фетчера, выполняющего вход в приложение:

```
import { useFetchers } from 'react-router-dom';
export default function App() {
    const fetchers = useFetchers();
    const loadFetcher = fetchers.find(
        fetcher => fetcher.formAction == '/login'
    );
    const isLoadFetcherIdle =
                loadFetcher ? loadFetcher.state == 'idle' : true;
    . . .
}
```

12.2. Поддержка асинхронных операций в React Router

React Router предоставляет инструменты, позволяющие приостановить вывод целевого компонента до момента получения результата асинхронной операции и, возможно, вывести в момент ожидания какое-либо другое содержимое. Они дополняют и расширяют аналогичные инструменты, предлагаемые фреймворком React (см. *разд. 9.2*).

Для использования этих инструментов требуется выполнить следующие действия:

1. В загрузчике, выдающем результат выполнения асинхронной операции, — вернуть результат вызова функции `defer()`:

   `defer(<простой объект с данными, передаваемыми компоненту>)`

   Отдельные значения, передаваемые целевому компоненту, должны храниться в отдельных свойствах задаваемого *простого объекта*. Это могут быть как непосредственно необходимые значения, так и содержащие их промисы (если эти значения генерируются асинхронно).

   Перечисленные далее действия выполняются в целевом компоненте.

2. Получить отправленный загрузчиком простой объект — вызовом хука `useLoaderData()` (см. *разд. 12.1.1.3*).

   Обычные значения, хранящиеся в полученном объекте, можно использовать сразу же.

3. Вставить в то место содержимого компонента, в котором должно выводиться значение из полученного промиса, компонент-приостановщик `Await`. В пропе `resolve` этого компонента нужно задать промис, хранящий выводимое значение.

4. Поместить в приостановщик `Await` функцию, в качестве единственного параметра принимающую извлеченное из промиса значение.

5. Поместить JSX-код, выводящий значение из промиса, в тело этой функции.

Пример:

```
// Модуль src\api.js;
import { defer } from 'react-router-dom';
export function getTodo(key) {
    return defer({
        todoPromise: loadTodo(key);
        prefix: 'Дело: '
    });
}

// Модуль src\TodoDetail.js
import { useLoaderData, Await } from 'react-router-dom';

export default function TodoDetail() {
    const { todoPromise, prefix } = useLoaderData();

    return (
        <Await resolve={todoPromise}>
            {todo => (
                <section>
                    <h1>{prefix} {todo.title}</h1>
                    <p>{todo.desc}</p>
                </section>
            )}
```

```
        </Await>
    );
}
```

Если значение, хранящееся в промисе, выводится в дочернем компоненте, в приостановщик `Await` можно поместить непосредственно этот компонент, а для получения выводимого значения из промиса в коде дочернего компонента вызвать хук `useAsyncValue()` без параметров. Пример:

```
import { useAsyncValue } from 'react-router-dom';

export default function TodoDetail() {
    . . .
    return (
        <Await resolve={todoPromise}>
            <__TodoDetail />
        </Await>
    );
}

function __TodoDetail() {
    const todo = useAsyncValue();

    return (
        <section>
            <h1>{todo.title}</h1>
            <p>{todo.desc}</p>
        </section>
    );
}
```

Приостановщик `Await` можно заключить в приостановщик `Suspense` (см. *разд. 9.2.1*), в котором задать содержимое, выводящееся во время ожидания завершения асинхронной операции:

```
import { Suspense } from 'react';
export default function TodoDetail() {
    . . .
    return (
        <Suspense fallback={<p>Идет загрузка...</p>}
            <Await resolve={todoPromise}>
                . . .
            </Await>
        </Suspense>
    );
}
```

12.3. Получение сведений о совпавших маршрутах и самом маршрутизаторе

12.3.1. Получение сведений о совпавших маршрутах

Этот программный инструмент пригодится, если используется иерархический список маршрутов (см. *разд. 11.2.2.2*) и в компоненте из родительского маршрута требуется получить данные из дочерних маршрутов (например, URL-параметры или данные, выданные загрузчиками).

Сведения о совпавших маршрутах возвращает хук `useMatches()`, вызванный без параметров.

Возвращенные этим хуком сведения представляют собой массив. Его элементы хранят сведения обо всех совпавших маршрутах, начиная с наиболее «внешнего» и заканчивая самым «глубоко» вложенным. Так, последний элемент массива хранит сведения о совпавшем конечном маршруте, предпоследний — о совпавшем узловом маршруте, в который вложен совпавший конечный маршрут, и т. д.

Каждый элемент возвращенного массива представляет собой простой объект со следующими свойствами:

- `id` — уникальный идентификатор маршрута в виде строки. В его качестве используется одно из двух:
  - якорь — если он задан в пропе `id` компонента маршрута;
  - комбинация из порядковых номеров маршрутов, начиная с наиболее «внешнего» и заканчивая самым «глубоко» вложенным, через дефисы. Номера начинаются с `0`;
- `pathname` — часть текущего пути, совпавшая с шаблонным путем;
- `data` — результат, выданный загрузчиком, или `undefined`, если загрузчик в маршруте не задан;
- `params` — простой объект с URL-параметрами, аналогичный выдаваемому хуком `useParams()` (см. *разд. 11.4.1*);
- `handle` — значение, указанное в пропе `handle` маршрута.

Вот пример получения в компоненте `App`, указанном в узловом маршруте, значения URL-параметра `key` из конечного маршрута с якорем `detail`:

```
import { useMatches } from 'react-router-dom';
export default function App() {
    const matches = useMatches();
    const detailMatch = matches.find(match => match.id == 'detail');
    const key = detailMatch?.params.key;
    . . .
}
```

12.3.2. Получение сведений о самом маршрутизаторе

Есть возможность получить сведения о самом маршрутизаторе: его состояние (простаивает или выполняет навигацию), текущее местоположение, пересылаемые данные и др.

Такие сведения возвращаются хуком `useNavigation()`, вызванным без параметров.

Возвращенные сведения представляются простым объектом со свойствами:

- `state` — обозначение состояния маршрутизатора в виде строки: `'idle'` (простаивает), `'loading'` (выполняет переход на другой экран) или `'submitting'` (отправляет данные воздействию).

  При обычной навигации на другой экран последовательность смены состояний такова:

  ```
  'idle' → 'loading' → 'idle'
  ```

  При отправке данных из веб-формы последовательность следующая:

  ```
  'idle' → 'submitting' → 'loading' → 'idle'
  ```

- `location` — местоположение, на которое выполняется переход, в виде простого объекта, аналогичного возвращаемому хуком `useLocation()` (см. *разд. 11.4.3*);
- `formData` — данные, занесенные в веб-форму, в виде объекта класса `FormData`, если при отправке данных была использована кодировка `application/x-www-form-urlencoded`. Если данные не отправлялись, хранит пустой объект класса `FormData`. Если использовалась другая кодировка, хранит значение `undefined`;
- `json` — данные, занесенные в веб-форму, в виде простого объекта. Доступны, если при отправке данных была использована кодировка `application/json`. Если данные не отправлялись, хранит пустой простой объект. Если использовалась другая кодировка, хранит значение `undefined`;
- `text` — данные, занесенные в веб-форму, в виде строки. Доступны, если при отправке данных была использована кодировка `text/plain`. Если данные не отправлялись, хранит пустую строку. Если использовалась другая кодировка, хранит значение `undefined`;
- `formAction` — путь, по которому были отправлены данные, в виде строки, если использовались компонент `Form` или хук `useSubmit()`. В остальных случаях — `undefined`;
- `formMethod` — обозначение HTTP-метода, посредством которого были отправлены данные, в виде строки, если использовались компонент `Form` или хук `useSubmit()`. В остальных случаях — `undefined`;
- `formEncType` — обозначение кодировки, в которой были отправлены данные, в виде строки, если использовались компонент `Form` или хук `useSubmit()`. В остальных случаях — `undefined`.

Вот так можно сделать интерфейс приложения полупрозрачным во время навигации:

```
import { useNavigation } from 'react-router-dom';
export default function App() {
    . . .
    const { state } = useNavigation();
    . . .
    return (
        <div className="container"
             style={{ opacity: state == 'idle' ? 1 : 0.2 }}>
            . . .
        </div>
    );
}
```

12.4. Передача произвольного значения целевому компоненту через маршрут

В *разд. 11.3.1.1*, *11.3.2* и *11.4.3* говорилось, что целевому компоненту можно отправить произвольное значение любого типа, указав его в пропе `state` гиперссылки или одноименной настройке перенаправления.

Существует еще один способ передать какое-либо значение целевому компоненту — записав его в пропе `handle` маршрута. Это значение будет доступно в свойстве `handle` простого объекта, находящегося в массиве, который возвращается хуком `useMatches()` (см. *разд. 12.3.1*).

Чаще всего таким образом компоненту передают часть функциональности, реализованной в виде функции или простого объекта с методами.

В листинге 12.1 приведен код компонента `Breadcrumbs`, создающего так называемые *хлебные крошки* — набор гиперссылок для перехода на последовательно вложенные друг в друга разделы приложения. Этот компонент извлекает простые объекты, содержащиеся в пропах `handle` маршрутов, и вызывает методы `crumb()` этих объектов, чтобы получить гиперссылки, которые и сформируют хлебные крошки.

Листинг 12.1. Код компонента `Breadcrumbs`

```
import { useMatches } from 'react-router-dom';

export default function Breadcrumbs() {
    const matches = useMatches();
    const crumbs = matches
                   .filter(match => Boolean(match.handle?.crumb))
                   .map(match => match.handle.crumb(match.data));
    const lastCrumb = crumbs[crumbs.length - 1];

    return (
        <nav className="breadcrumb">
```

```
            <ul>
                {crumbs.map((crumb, index) => (
                    <li key={index}
                     className={(crumb == lastCrumb ? 'is-active' : '')}>
                        {crumb}
                    </li>
                ))}
            </ul>
        </nav>
    );
}
```

Пример указания в маршрутах объектов с методами, которые создадут отдельные гиперссылки:

```
import { Route, Link } from "react-router-dom";
const router = createBrowserRouter(
    createRoutesFromElements(
        <Route path="/" element={<App />}
               handle={{
                   crumb: () => <Link to="/">Главная</Link>
               }}
        >
            <Route index={true} element={<TodoList />} />
            <Route path=":key" element={<TodoDetail />}
                   handle={{
                       crumb: todo => <Link to="#">{todo.title}</Link>
                   }} />
            . . .
        </Route>
    )
);
```

Использование компонента `Breadcrumbs`:

```
import Breadcrumbs from './Breadcrumbs.js';
export default function TodoDetail() {
    . . .
    return (
        <section>
            <Breadcrumbs />
            . . .
        </section>
    );
}
```

12.5. Сохранение позиции прокрутки

Если в приложении списка дел (см. *главы 1* и *2*) мы прокрутим длинный перечень дел в самый низ, щелкнем на каком-либо деле, чтобы открыть экран со сведениями о нем, и вернемся на перечень дел нажатием кнопки **Назад**, то увидим его начало. Позиция, на которую был прокручен перечень дел, после перехода на другой экран и последующего возврата не сохранилась.

Сохранить позицию прокрутки при навигации с использованием кнопок **Назад** и **Вперед** поможет компонент `ScrollRestoration`. Он ставится в содержимом разметочного компонента, записанного в самом «внешнем» из узловых маршрутов. Содержимое в этом компоненте не указывается.

Пример:

```
import { ScrollRestoration } from 'react-router-dom';
export default function App() {
    . . .
    return (
        <div className="container">
            . . .
            <ScrollRestoration />
        </div>
    );
}
```

Позиция прокрутки у каждого экрана сохраняется в особом массиве и получает уникальный ключ, по которому компонент `ScrollRestoration` ее впоследствии извлекает, чтобы применить. Этот ключ вычисляется на основе текущего местоположения и сведений о совпавших маршрутах.

При навигации на другой экран компонент `ScrollRestoration` вычисляет ключ и ищет в массиве элемент с этим ключом. Если таковой находится, компонент извлекает его и использует для установки позиции прокрутки. В противном случае создается новый элемент массива, и в нем сохраняется позиция прокрутки, равная `0`.

Компонент `ScrollRestoration` поддерживает проп `getKey`, в котором можно указать функцию, генерирующую уникальный ключ для каждого экрана. Эта функция должна принимать два параметра:

- текущее местоположение в виде простого объекта, аналогичного возвращаемому хуком `useLocation()` (см. *разд. 11.4.3*);
- массив со сведениями обо всех совпавших маршрутах, подобный возвращаемому хуком `useMatches()` (см. *разд. 12.3.1*).

Возвращать функция должна сгенерированный ею уникальный ключ.

По умолчанию в качестве ключа используется уникальный идентификатор, хранящийся в свойстве `key` объекта местоположения (подробности о нем — в *разд. 11.4.3*). Такой ключ привязан к текущему объекту местоположения, а не к экрану, вследст-

вие чего компонент `ScrollRestoration` в изначальном состоянии гарантирует сохранение позиции прокрутки лишь при навигации кнопками **Назад** и **Вперед**.

Чтобы компонент `ScrollRestoration` сохранял позицию прокрутки и при навигации щелчками на гиперссылках, ключ для элементов массива следует генерировать на основе каких-либо значений, привязанных к определенным экранам, а не к объектам местоположения. Таким значением может выступить, например, путь.

Вот пример указания в компоненте функции, выдающей путь в качестве ключа:

```
<ScrollRestoration getKey={location => location.pathname} />
```

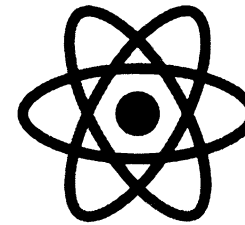
Как говорилось ранее, при навигации на другой экран, если его позиция прокрутки не была сохранена ранее в массиве, в нем создается новый элемент с позицией прокрутки, равной `0`. Соответственно, позиция прокрутки целевого экрана после перехода на него также будет равной `0`.

Чтобы React Router после перехода на другой экран сохранял текущую позицию прокрутки, следует в компоненте гиперссылки, ведущей на этот экран, поместить проп `preventScrollReset` со значением `true`.

Пример:

```
<Link to={`/${item.key}`} preventScrollReset>[item.title}</Link>
```

ГЛАВА 13



Преобразователи данных React

Программный код, изменяющий и обновляющий состояние компонента, может быть очень сложным и громоздким (особенно если состояние хранит массив или объект, элементы или свойства которого следует изменять). Естественно, возникает желание вынести такой код за пределы компонента — для упрощения отладки и последующего сопровождения.

Разумеется, этот код можно оформить в виде обычных функций. Однако React предлагает более удобную альтернативу — преобразователи.

Преобразователь React[1] — это программный код, принимающий от компонентов предписания на выполнение тех или иных действий над состоянием и на их основе выполняющий эти действия. Предписания от компонентов, получаемые им, носят название *влияний*[2].

13.1. Влияния

Если в том или ином компоненте планируется использовать преобразователь React, то перед программированием этого компонента следует составить список влияний, поддерживаемых будущим преобразователем. Также надо продумать структуру влияний и данные, которые будут пересылаться с их помощью.

Этот шаг очень важен, поскольку именно на основе списка и структуры влияний впоследствии будут программироваться преобразователи.

Влияние — предписание преобразователю на выполнение того или иного действия над состоянием компонента — представляет собой простой объект, содержащий следующие свойства:

---

[1] В документации по React — reducer.

[2] В документации по React — actions. Точно так же называются воздействия, описанные в *разд. 8.2.2*. Автор книги дал им разные названия — во избежание путаницы.

- `type` — наименование действия, производимого над данными, в виде строки.

  Может быть выбрано произвольно. Сами разработчики React в документации по фреймворку применяют наименования формата:

  <глагол в прошедшем времени>_<существительное>

  тем самым как бы сообщая, что преобразователь сделал с состоянием.

  Примеры: `added_todo` (добавлено дело), `deleted_todo` (удалено дело).

  `type` — единственное свойство, которое обязательно должно присутствовать в любом влиянии;

- `payload` — данные, необходимые для выполнения действия (например, простой объект, хранящий добавляемое дело);
- любые другие свойства, хранящие произвольные дополнительные данные.

Вот пример влияния, предписывающего добавить в список новое дело:

```
{
    type: 'added_todo',
    payload: {
        title: 'Написать первое React-приложение',
        desc: 'Список запланированных дел',
        image: '',
        done: false,
        createdAt: '19.07.2024, 14:47:00',
        key: 1721389620000
    }
}
```

13.2. Создание преобразователя React

Преобразователь React — это код, который вносит в указанное состояние изменения, заданные в полученном им влиянии.

13.2.1. Написание преобразователя React

Преобразователь реализуется в виде функции, принимающей два параметра:

- текущее значение состояния, обрабатываемого преобразователем;
- влияние, переданное преобразователю компонентом.

В качестве результата преобразователь должен вернуть новое значение состояния.

Если преобразователь получил влияние, предписывающее выполнить действие, которое он не поддерживает, он должен вернуть в качестве результата текущее состояние.

В листинге 13.1 приведен код преобразователя, обрабатывающего список дел.

Листинг 13.1. Код преобразователя, работающего со списком дел

```
export default function reducer(state, action) {
    let newTodos, deed;
    switch(action.type) { // ..................................... 1
        case 'added_todo':
            return [...state, action.payload];
        case 'set_done_todo':
            newTodos = [...state];
            deed = newTodos.find(
                current => current.key === action.payload
            );
            if (deed)
                deed.done = true;
            return newTodos;
        case 'deleted_todo':
            newTodos = state.filter(
                current => current.key !== action.payload
            );
            return newTodos;
        default: // ..................................... 2
            return state;
    }
}
```

В коде преобразователя извлекаем из свойства `type` полученного влияния наименование требуемой манипуляции с данными и, в зависимости от этого наименования, выполняем ту или иную манипуляцию. Для этого используем выражение выбора (поз. 1 в листинге 13.1) — такова общепринятая практика.

Если преобразователь получит влияние с неподдерживаемым наименованием действия, не забываем вернуть текущее состояние (поз. 2).

Если преобразователь используется совместно несколькими компонентами, его код имеет смысл вынести в отдельный файл, которому можно дать произвольное имя.

13.2.2. Регистрация преобразователя React

Написанный преобразователь следует зарегистрировать, одновременно создав состояние, с которым он должен работать.

Регистрацию заданного *преобразователя* и создание состояния с занесением в него указанного *начального значения* выполняет хук `useReducer()`:

```
useReducer(<преобразователь>, <начальное значение состояния>[,
                               <инициализирующая функция>])
```

- Если *инициализирующая функция* *не* задана — в созданное состояние будет занесено *начальное значение*;

- если *инициализирующая функция* задана — она будет вызвана с передачей ей в качестве единственного параметра указанного *начального значения*, и в созданное состояние будет занесен возвращенный *функцией* результат.

Хук возвращает массив из двух элементов:

- текущего значения состояния;
- *функции-диспетчера* — выполняющей отправку влияния преобразователю.

Практически всегда результат, возвращенный хуком, подвергается деструктурированию, и его элементы присваиваются двум константам. Второй константе, хранящей функцию-диспетчер, обычно дают имя, составленное из префикса `dispatch` и имени первой константы.

Пример (предполагается, что код преобразователя хранится в модуле src\reducer.js):

```
import { useReducer } from 'react';
import reducer from './reducer.js';
import initialTodos from './todos.js';

export default function App() {
    const [todos, dispatchTodos] = useReducer(reducer, initialTodos);

    return (
        <div className="container">
            <TodoList list={todos} dispatch={dispatchTodos} />
            <TodoAdd dispatch={dispatchTodos} />
        </div>
    );
}
```

13.3. Работа с преобразователями React

Чтобы внести какое-либо изменение в состояние, обрабатываемое преобразователем, следует отправить этому преобразователю соответствующим образом оформленное влияние.

Отправка влияний выполняется функцией-диспетчером, возвращенной хуком `useReducer()` во втором элементе массива (см. *разд. 13.2.2*). Функция-диспетчер имеет следующий формат вызова:

<функция-диспетчер>(*<отправляемое влияние>*)

Результата она не возвращает.

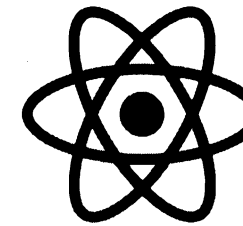
Примеры:

```
// Модуль src\TodoList.js
export default function TodoList({ list, dispatch }) {
    return (
        <table>
```

```
            {list.map(item => (
                . . .
                <button onClick={() => {
                    dispatch({
                        type: 'set_done_todo',
                        payload: item.key
                    })}
                }>
                    Выполнено
                </button>
                . . .
                <button onClick={() => {
                    dispatch({
                        type: 'deleted_todo',
                        payload: item.key
                    })}
                }>
                    Удалить
                </button>
                . . .
            ))}
        </table>
    );
}

// Модуль src\TodoAdd.js
export default function TodoAdd({ dispatch }) {
    . . .
    const handleFormSubmit = evt => {
        evt.preventDefault();
        const newDeed = {title, desc, image, done: false};
        . . .
        dispatch({ type: 'added_todo', payload: newDeed });
        evt.target.reset();
    };
    . . .
}
```

ГЛАВА 14



Библиотека Redux: централизованное хранение состояния, раздел 1

Данные, обрабатываемые React-приложением, часто хранятся в состоянии одного из компонентов — обычно компонента приложения. Там же содержится логика, обрабатывающая эти данные. Именно так хранение и обработка данных реализованы в первой версии нашего приложения, написанной в *главе 1*.

Такой подход имеет два преимущества. Во-первых, все данные и обрабатывающая их логика находятся в одном месте. Во-вторых, при изменении данных изменяется состояние компонента, и последний, вместе со своими потомками, автоматически обновляется. И то и другое существенно упрощает программирование.

Однако в сложных приложениях, манипулирующих объемными, сложно структурированными данными, этот подход может оказаться неприемлемым. Прежде всего из-за того, что компонент, в котором хранятся данные, оказывается слишком сложным, «перегруженным» программной логикой. А разделение данных и логики между несколькими компонентами лишь дополнительно усложнит разработку.

Выходом может оказаться вынос обрабатываемых данных и обрабатывающей их логики в отдельную сущность. Сделать это поможет семейство дополнительных библиотек, главной из которых является Redux.

14.1. Библиотека Redux

Redux — это дополнительная библиотека, реализующая хранение обрабатываемых приложением данных в отдельной сущности, называемой *хранилищем*[1], а программной логики, обрабатывающей эти данные, — в другой сущности, называемой *преобразователем Redux*[2].

Хранилище Redux позволяет хранить всего одно значение, называемое *состоянием хранилища*. Это значение может быть любого типа. Однако практически всегда

---

[1] В оригинальной документации — store.

[2] Не путать с преобразователем React (см. *главу 13*).

в хранилище записывают объект, свойства которого и хранят нужные приложению данные.

> ***Внимание!***
>
> В хранилище следует хранить лишь данные, используемые несколькими компонентами (например, список запланированных дел). Служебные данные, принадлежащие какому-либо одному компоненту (наподобие признака, выводить на страницу меню навигации или нет), лучше сохранить в состоянии этого компонента.

Компонент, желающий выполнить определенное действие над данными, находящимися в хранилище, посылает ему соответствующее влияние (как и в случае преобразователя React — см. *главу 13*). В составе влияния он указывает наименование действия, которое следует произвести над данными (скажем, добавить в список новое дело), и какие-либо дополнительные сведения (например, объект с новым делом, которое нужно добавить в список). Получив влияние, хранилище передает его преобразователю, который и выполняет необходимые манипуляции.

Компонент, желающий получать оповещения об изменении состояния хранилища, регистрирует в нем *слушатель*. Последний выполняется после изменения состояния хранилища и производит заданные разработчиком действия.

Селектор — программная сущность, которая извлекает из хранилища заданное значение. Селекторы активно применяются в слушателях для упрощения программирования.

Для установки библиотеки Redux следует набрать команду:

```
npm install redux
```

> ***На заметку***
>
> Полное описание библиотеки Redux приведено по интернет-адресу: **https://redux.js.org/**.
>
> Redux никак не привязана к React и может использоваться с любым другим фреймворком, равно как и вообще без каких-либо фреймворков.

14.1.1. Влияния

Влияния в Redux практически ничем не отличаются от влияний в React (см. *разд. 13.1*). Они также представляются простыми объектами JavaScript, которые содержат по меньшей мере свойства `type` (наименование производимого действия) и `payload` (данные, необходимые для выполнения этого действия).

Единственное отличие касается наименований действий, которые, согласно рекомендациям разработчиков Redux, должны удовлетворять формату:

<категория>/<собственно действие>

Категория объединяет несколько действий, выполняющих манипуляции над одним массивом данных. Так, для действия, создающего новое дело, можно указать наименование `todos/add`, для пометки дела как выполненного — `todos/setDone`, а для регистрации нового посетителя — `users/register`.

14.1.2. Преобразователь Redux

Преобразователь Redux вызывается хранилищем при получении очередного влияния от одного из компонентов.

Преобразователь реализуется в виде функции, принимающей два параметра:

- текущее состояние хранилища.

  Это необязательный параметр. В качестве значения по умолчанию ему можно указать изначальное состояние, которое следует записать в хранилище при его создании. Это изначальное состояние должно быть представлено в виде простого объекта.

  Кроме того, изначальное состояние можно задать при создании хранилища (рассмотрено в *разд. 14.1.3*);

- влияние, полученное хранилищем.

В качестве результата преобразователь должен вернуть копию состояния с изменениями, внесенными в нее согласно полученному влиянию.

> ***Внимание!***
>
> Вносить изменения в текущее состояние хранилища и возвращать его в качестве результата нельзя — в этом случае Redux не сможет отследить изменения в состоянии хранилища и не проинформирует о них компоненты.

Если преобразователь получил влияние, предписывающее выполнить действие, которое он не поддерживает, он должен вернуть в качестве результата текущее состояние хранилища.

Как правило, преобразователю дают имя, заканчивающееся суффиксом `reducer`, — согласно действующим соглашениям. Его код обычно сохраняют в отдельном модуле.

В листинге 14.1 приведен код преобразователя, обрабатывающего список дел. В целом он аналогичен преобразователю из листинга 13.1.

Листинг 14.1. Преобразователь, обрабатывающий список дел

```
export default function todoReducer(state, action) {
    let newState, deed;
    switch(action.type) {
        case 'todos/add':
            return { ...state, todos: [...state.todos, action.payload] };
        case 'todos/setDone':
            newState = { ...state };
            deed = newState.todos.find(
                current => current.key === action.payload
            );
            if (deed)
                deed.done = true;
            return newState;
```

```
        case 'todos/delete':
            newState = { ...state };
            newState.todos = state.todos.filter(
                current => current.key !== action.payload
            );
            return newState;
        default:
            return state;
    }
}
```

14.1.3. Хранилище

Для создания нового хранилища Redux применяется функция `createStore()` из модуля `redux`:

```
createStore(<преобразователь>[, <изначальное состояние>=undefined[,
                                <расширитель>=undefined]])
```

Единственное, что требуется указать при создании хранилища, — работающий с ним *преобразователь*.

Также можно задать *изначальное состояние*, которое будет записано в хранилище при его создании. *Изначальное состояние* должно быть задано в виде простого объекта.

Что такое *расширитель*, мы узнаем позже.

Функция `createStore()` возвращает объект, представляющий созданное хранилище. Этот объект следует сохранить в какой-либо переменной.

Как правило, код, создающий хранилище, выносится в отдельный модуль, обычно с именем src\store.js. Переменную, содержащую объект хранилища, в этом случае следует экспортировать — она понадобится в других модулях.

Вот пример кода, создающего хранилище и записывающего в него в качестве изначального состояния массив из двух планируемых дел (предполагается, что преобразователь хранится в модуле src\reducer.js):

```
import { createStore } from 'redux';
import todoReducer from './reducer.js';

const date1 = new Date(2024, 6, 18, 10, 24);
const date2 = new Date(2024, 6, 19, 14, 47);

const initialData = {
    todos: [
        {
            title: 'Изучить React',
            . . .
            key: date1.getTime()
        },
```

```
        {
            title: 'Написать первое React-приложение',
            . . .
            key: date2.getTime()
        }
    ]
};

const store = createStore(todoReducer, initialData);
export default store;
```

Хранилище следует создавать перед созданием всех компонентов, содержащихся в проекте, включая компонент приложения. Наилучшее место для этого — стартовый модуль src\index.js. Выражение, импортирующее модуль src\store.js, в котором создается хранилище, ставится перед выражением импорта компонента приложения, например:

```
import './store.js';
import App from './App.js';
```

14.1.4. Селекторы

Согласно принципам Redux компоненты не должны обращаться к хранилищу за данными самостоятельно. Вместо этого они должны вызывать функции-селекторы, выдающие нужные данные.

Селектор Redux — это обычная функция, принимающая следующие параметры:

- текущее состояние хранилища — это единственный обязательный параметр;
- любые другие значения, необходимые для извлечения данных (например, идентификатор выводимого дела).

В качестве результата селектор должен возвращать требуемое значение, извлеченное из состояния хранилища.

По действующему соглашению имя селектора должно предваряться префиксом `select`.

Код селекторов также сохраняется в отдельном модуле, обычно это src\selectors.js.

Пример простейшего селектора, выдающего массив дел:

```
export function selectTodos(state) {
    return state.todos;
}
```

Пример селектора, выдающего дело с указанным идентификатором:

```
export const selectTodo = (state, key) => {
    return state.todos.find(current => current.key === key);
}
```

14.1.5. Получение данных из хранилища

Получить данные из хранилища проще всего — достаточно вызвать нужный селектор, передав ему в первом параметре объект состояния хранилища.

Объект состояния хранилища можно получить, вызвав у хранилища метод `getState()`.

Пример получения массива дел:

```
import store from './store.js';
import { selectTodos } from './selectors.js';

export default function TodoList() {
    const state = store.getState();
    const list = selectTodos(state);

    return (
        <table>
            {list.map(item => (
                . . .
            ))}
        </table>
    );
}
```

Пример получения дела, чей идентификатор передан с URL-параметром `key`:

```
import store from './store.js';
import { selectTodo } from './selectors.js';

export default function TodoDetail() {
    const { key } = useParams();
    const deed = selectTodo(store.getState(), key);

    return (
        <section>
            <h1>{deed.title}</h1>
            . . .
        </section>
    );
}
```

14.1.6. Отправка влияний

Чтобы внести какие-либо изменения в данные, находящиеся в хранилище, следует отправить этому хранилищу соответствующее влияние.

Для отправки хранилищу заданного *влияния* у объекта хранилища нужно вызвать метод `dispatch(<влияние>)`. Метод вернет в качестве результата переданное ему влияние. Этот результат малополезен, поэтому его обычно игнорируют.

Вот пример кода, отправляющего влияния «пометить дело как выполненное» и «удалить дело» по нажатию соответствующих кнопок:

```
export default function TodoList() {
    . . .
    const handleSetDoneClick = key => {
        store.dispatch({ type: 'todos/setDone', payload: key });
    }
    const handleDeleteClick = key => {
        store.dispatch({ type: 'todos/delete', payload: key });
    }

    return (
        <table>
            {list.map(item => (
                . . .
                <button title="Пометить как выполненное"
                        onClick={() => {handleSetDoneClick(item.key)}}>
                    &#9745;
                </button>
                . . .
                <button title="Удалить"
                        onClick={() => {handleDeleteClick(item.key)}}>
                    &#9746;
                </button>
                . . .
            ))}
        </table>
    );
}
```

14.1.7. Регистрация слушателей

Чтобы компонент смог получать оповещения об изменениях в данных, находящихся в хранилище, он должен зарегистрировать *слушатель*[1].

Слушатель реализуется в виде обычной функции. Она не должна ни принимать параметров, ни возвращать результат.

В теле слушателя обычно выполняют принудительное обновление компонента. Сделать это можно, задав у какого-либо из состояний компонента новое значение. Можно даже заранее создать какое-либо состояние с логическим значением и в теле слушателя изменять его значение на противоположное (с `false` на `true` и наоборот).

Регистрация заданного *слушателя* выполняется вызовом у объекта хранилища метода `subscribe(<`*слушатель*`>)`.

[1] В оригинальной документации — listener.

В качестве результата метод возвращает сгенерированную Redux функцию, вызовом которой можно отменить регистрацию этого слушателя.

Перед демонтированием компонента следует отменить регистрацию слушателя. Это можно сделать, вызвав функцию, возвращенную методом `subscribe()`.

Вот пример регистрации слушателя в компоненте `TodoList`:

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

export default function TodoList() {
    . . .
    const [, setFlag] = useState(false);

    const refresh = () => {setFlag(flag => !flag)};

    useEffect(() => {
        const unsubscribe = store.subscribe(refresh);
        return () => {unsubscribe()};
    }, []);
    . . .
}
```

14.1.8. Создатели влияний

Создатель влияний[1] — это функция, принимающая ряд параметров и возвращающая созданный на их основе объект влияния определенного типа. Применение создателей влияний позволяет немного упростить код, отправляющий влияния хранилищу, особенно если влияния отправляются в нескольких местах кода.

Как правило, код создателей влияний выносится в отдельный модуль с именем, например src\actionCreators.js.

Пример создателя влияний, помечающих дело как выполненное:

```
export function setDoneTodo(key) {
    return { type: 'todos/setDone', payload: key };
}
```

Пример использования этого создателя влияний:

```
import { setDoneTodo } from './actionCreators.js';
export default function TodoList() {
    . . .
    const handleSetDoneClick = key => {
        store.dispatch(setDoneTodo(key));
    }
    . . .
}
```

[1] В документации — action creator.

14.1.9. Отправители влияний

Еще значительнее упростить код можно, преобразовав написанные ранее создатели влияний в *отправители влияний*[1]. Они не только создают влияния на основе полученных параметров, но и сами отправляют их хранилищу.

Для преобразования создателя влияний в их отправителя следует использовать функцию `bindActionCreators()` из модуля `redux`:

```
bindActionCreators(<создатель влияний>|<набор создателей влияний>,
                   <метод dispatch()>)
```

Если первым параметром указать *создатель влияний*, функция вернет созданный на его основе отправитель влияний.

Пример кода, создающего отправитель влияний, который помечает дело как выполненное:

```
import { bindActionCreators } from 'redux';
import store from './store.js';
function __setDoneTodo(key) {
    return { type: 'todos/setDone', payload: key };
}
export const setDoneTodo = bindActionCreators(__setDoneTodo, store.dispatch);
```

Пример использования этого отправителя влияний:

```
import { setDoneTodo } from './actionCreators.js';
export default function TodoList(props) {
    . . .
    const handleSetDoneClick = key => {setDoneTodo(key)};
    . . .
}
```

Отправитель влияний можно использовать и в обработчике события:

```
<button title="Пометить как выполненное"
        onClick={() => {setDoneTodo(item.key)}}>
    &#9745;
</button>
```

Первым параметром функции `bindActionCreators()` также можно передать *набор создателей влияний* — простой объект, свойства которого хранят отдельные создатели влияний, подлежащие преобразованию. В таком случае функция вернет тот же объект, свойства которого будут хранить отправители влияний, созданные на основе указанных создателей. Пример:

```
// Модуль src\actionCreators.js
export const boundActionCreators = bindActionCreators({
    setDoneTodo: setDoneTodoUnbound,
    . . .
}, store.dispatch);
```

---

[1] В документации — bound action creator.

```
// Модуль src\TodoList.js
import { boundActionCreators } from './actionCreators.js';
export default function TodoList(props) {
    . . .
    const handleSetDoneClick = key => {
        boundActionCreators.setDoneTodo(key);
    }
    . . .
}
```

14.1.10. Специфические случаи применения Redux

14.1.10.1. Составные преобразователи

В больших приложениях, выполняющих множество действий над данными, код единственного преобразователя может стать слишком объемным, что затруднит программирование и сопровождение. В этом случае удобнее разбить его на несколько преобразователей и свести их в один — *составной*, — который и передать функции `createStore()`, создающей хранилище (см. *разд. 14.1.3*).

Каждый из отдельных преобразователей, входящих в составной преобразователь, станет манипулировать одной из частей данных, записанных в хранилище, — *срезом* (отчего и носит название *преобразователя среза*). Например, один преобразователь среза будет работать со списком дел, а другой — со сведениями о текущем посетителе.

Срез представляет собой значение произвольного типа (например, массив или объект), записанный в одно из свойств состояния хранилища. Имена свойств, в которые будут записаны срезы, можно выбрать произвольно.

В качестве первого параметра преобразователю среза передается *содержимое обрабатываемого им среза* (а не все состояние хранилища, как у обычного преобразователя). Это следует учитывать при написании кода такого преобразователя.

У первого параметра преобразователя среза необходимо указать в качестве значения по умолчанию изначальные данные, которые станут содержимым среза. Задавать их в вызове функции `createStore()` не нужно.

Для создания составного преобразователя служит функция `combineReducers()` из модуля `redux`:

```
combineReducers(<набор преобразователей>)
```

Набор преобразователей должен представлять собой простой объект, свойства которого хранят отдельные преобразователи, — на их основе и создается составной преобразователь. Свойства *набора* зададут срезы хранилища, в которых будут храниться данные, обрабатываемые соответствующими преобразователями.

Функция возвращает готовый к использованию составной преобразователь — его и следует передать функции `createStore()` с первым параметром.

Пример:

```
import { combineReducers } from 'redux';

const date1 = new Date(2024, 6, 18, 10, 24);
const date2 = new Date(2024, 6, 19, 14, 47);

const initialTodos = [
    {
        title: 'Изучить React',
        . . .
        key: date1.getTime()
    },
    {
        title: 'Написать первое React-приложение',
        . . .
        key: date2.getTime()
    }
];

function todosReducer(state=initialTodos, action) {
    let newTodos, deed;
    switch(action.type) {
        case 'todos/add':
            return [...state, action.payload];
        . . .
        default:
            return state;
    }
}

function usersReducer(state={ currentUser: null }, action) {
    switch(action.type) {
        case 'users/login':
            return { currentUser: action.payload };
        case 'users/logout':
            return { currentUser: null };
        default:
            return state;
    }
}

const reducer = combineReducers({
    data: todosReducer,
    user: usersReducer
});
export default reducer;
```

Влияния отправляются так же, как и при использовании единственного преобразователя, — вызовом метода `dispatch()` у хранилища. Метод `getState()` хранилища будет возвращать, как и ранее, все состояние хранилища, а не данные среза. Так что код, обрабатывающий данные из хранилища, переписывать не придется.

14.1.10.2. Замена преобразователя

Redux позволяет заменить преобразователь, используемый хранилищем, на другой. Это может понадобиться, например, если один экран приложения обрабатывает данные одним способом, а другой — иным. В таком случае удобнее объявить два преобразователя и менять их при переходе с экрана на экран.

Замена преобразователя выполняется вызовом метода `replaceReducer()` хранилища:

```
replaceReducer(<новый преобразователь>)
```

Пример:

```
import store from './store.js';
import { newReducer } from './reducers.js';
. . .
store.replaceReducer(newReducer);
```

14.1.10.3. Мемоизация селекторов. Библиотека Reselect

Если какой-либо селектор выполняет длительные вычисления, имеет смысл вызывать его как можно реже, чтобы повысить производительность приложения. Достичь этого удается, мемоизировав селектор (о мемоизации рассказывалось в *разд. 4.3.2*).

Мемоизированный селектор будет вызван, только если значения указанных при его создании зависимостей изменились со времени его предыдущего вызова. В противоположном случае будет возвращен результат, вычисленный при предыдущем вызове селектора и сохраненный в памяти.

Для мемоизации селекторов применяется дополнительная библиотека *Reselect*. Для ее установки следует подать команду:

```
npm install reselect
```

> ***НА ЗАМЕТКУ***
>
> **Полное описание библиотеки Reselect можно найти по интернет-адресу: https://reselect.js.org/.**

Чтобы превратить обычный селектор в мемоизированный, достаточно использовать функцию `createSelector()` из модуля `reselect`. Она поддерживает два формата вызова:

```
createSelector(<функция зависимости 1>, <функция зависимости 2>, . . .,
               <функция зависимости n>, <преобразуемый селектор>)
createSelector(<массив функций зависимости>, <преобразуемый селектор>)
```

В первом формате *функции зависимостей* указываются непосредственно в параметрах функции `createSelector()`. Во втором формате они передаются в первом параметре в виде *массива*.

Функций зависимостей может быть произвольное количество. Каждая из них должна принимать параметры, передаваемые при вызове мемоизированному селектору, и возвращать результат, который станет одной из зависимостей формируемого мемоизированного селектора.

Преобразуемый селектор должен принимать в качестве параметров результаты, возвращенные всеми заданными *функциями зависимостей*. Возвращенный им результат и станет значением, которое будет выдаваться мемоизированным селектором.

При первом вызове мемоизированного селектора сначала будут вызваны все указанные *функции зависимостей*, и возвращенные ими результаты сохранятся в памяти. После этого выполнится сам *преобразуемый селектор*, который получит в качестве параметров значения, возвращенные *функциями зависимостей*. Результат, возвращенный *селектором*, будет выдан программному коду и также сохранен в памяти.

При последующих вызовах мемоизированного селектора также сначала будут вызваны *функции зависимостей*. Если хотя бы одна из них вернет результат, не совпадающий с результатом, возвращенным при ее предыдущем вызове, *селектор* будет выполнен. В противном случае мы получим результат предыдущего вызова *селектора*, сохраненный в памяти. Результаты, возвращенные *функциями зависимостей*, при этом также будут сохранены, заменив предыдущие.

Код, создающий мемоизированный селектор, должен находиться вне какого бы то ни было компонента.

Вот пример мемоизированного селектора, выдающего дело с указанным идентификатором и вызываемого повторно при изменении массива дел и (или) идентификатора:

```
import { createSelector } from 'reselect';
const selectItem = createSelector(
    state => state.todos,
    (state, key) => key,
    (todos, key) => {
        key = +key;
        return todos.find(current => current.key === key);
    }
);
```

Вызывается мемоизированный селектор так же, как и обычный:

```
export default function todoDetail({ key }) {
    const deed = selectItem(store.getState(), key);
    . . .
}
```

14.2. Выполнение действий с данными из внешних источников

Правила программирования с использованием Redux требуют, чтобы преобразователь работал только с данными, полученными в составе очередного влияния и находящимися в хранилище. Получать данные из внешнего источника (cookie, хранилища Web API, IndexedDB, бэкенда) и отправлять данные во внешний источник не допускается. В преобразователе также не допускается задействовать внешние программные инструменты (в особенности асинхронные): выводить сообщения в консоль веб-обозревателя, создавать таймеры, отправлять запросы бэкенду и т. п.

14.2.1. Использование посредников Redux

Однако подобного рода действия можно выполнять в *посредниках*[1], которые принимают влияния от компонентов до того, как они достигнут преобразователя, и могут:

- добавлять в них какие-либо значения (например, данные, загруженные с бэкенда посредством асинхронного запроса);
- заменять их другими влияниями;
- выполнять дополнительные операции (например, выводить сообщения в консоль);
- отклонять.

К хранилищу можно привязать произвольное количество посредников. Они будут вызываться последовательно, в том порядке, в котором привязаны.

Посредник Redux реализуется в виде обычной функции, которая должна принимать в качестве параметра объект хранилища. Этому параметру можно дать имя `store`.

Функция вернет вторую функцию, которая должна принимать в качестве параметра ссылку на следующий посредник в последовательности или на метод `dispatch()` хранилища, если это последний посредник. Этот параметр традиционно называется `next`.

Вторая функция вернет третью функцию, которая должна принимать в качестве параметра очередное влияние, отправленное хранилищу. Этому параметру обычно дают имя `action`.

В третьей функции и выполняется обработка поступающих влияний. Например, можно изменить значения свойств влияния, указав другие данные, предназначенные к обработке, или даже другой тип влияния.

Чтобы передать измененное (или оставшееся неизменным) влияние следующему посреднику или хранилищу (если текущий посредник — последний), в третьей функции следует вызвать функцию, полученную второй функцией с параметром `next`, передав ей влияние, полученное третьей функцией с параметром `action`.

[1] В документации — middleware.

Результат, возвращенный функцией из параметра `next`, следует вернуть из третьей функции.

В посреднике можно подавить дальнейшее прохождение текущего влияния и создать вместо него другое. Влияние в посреднике создается точно так же, как и в компоненте, — вызовом метода `dispatch()` хранилища, создателя или отправителя влияний. Как показывает практика, можно создать сразу несколько новых влияний — и все они будут обработаны. Никакого результата из третьей функции при этом возвращать не нужно.

Обычно код посредников выносят в отдельный модуль с именем src\middleware.js. Если посредников слишком много, их лучше разнести на несколько модулей.

Вот пример, демонстрирующий принципы написания посредников Redux:

```
function middleware(store) {
    return function (next) {
        return function (action) {
            switch (action.type) {
                case 'actions/changeData':
                    // Заменяем переданные во влиянии данные
                    action.payload = 123;
                    // Добавляем во влияние дополнительные данные
                    action.otherData = 'Firebase';
                    // Вызываем следующий посредник (или хранилище),
                    // передавая ему влияние, тем самым продолжая
                    // его обработку
                    return next(action);
                case 'actions/changeAction':
                    // Заменяем полученное влияние другим
                    store.dispatch({
                        type: 'actions/action123',
                        payload: 321
                    });
                    // Следующий посредник (или хранилище) не вызываем,
                    // тем самым прерывая обработку текущего влияния
                    return;
                case 'actions/logToConsole':
                    // Выводим данные в консоль
                    console.log(action.payload);
                    // Продолжаем обработку текущего влияния
                    return next(action);
                case 'actions/abort':
                    // Прерываем обработку этого влияния
                    return;
                default:
                    return next(action);
            }
        }
    }
}
```

На практике удобнее создать несколько более мелких посредников, каждый из которых обрабатывает влияния какого-либо одного типа.

Написанные посредники следует привязать к хранилищу. Для этого надо использовать функцию `applyMiddleware()` из модуля `redux`:

```
applyMiddleware(<посредник 1>, <посредник 2>, . . ., <посредник n>)
```

Посредники будут выполняться в том порядке, в котором они указаны в вызове функции, слева направо.

Функция возвращает объект-*расширитель хранилища*, добавляющий ему дополнительную функциональность. Этот объект следует передать функции `createStore()` третьим параметром (см. *разд. 14.1.3*).

В листинге 14.2 приведен код простого преобразователя, обрабатывающего влияния двух типов: `todos/setTodos` (заносит в хранилище список дел, хранящийся в свойстве `payload`) и `users/setUser` (заносит в хранилище объект текущего посетителя также из свойства `payload`). И список дел, и объект посетителя будут загружаться с бэкенда посредниками, которые мы напишем далее.

Листинг 14.2. Преобразователь, обрабатывающий список дел и сведения о текущем посетителе

```
export default function mainReducer(state, action) {
    switch(action.type) {
        case 'todos/setTodos':
            return { ...state, todos: action.payload };
        case 'users/setUser':
            return { ...state, currentUser: action.payload };
        default:
            return state;
    }
}
```

Листинг 14.3 демонстрирует код двух посредников. Посредник `userMiddleware()` обрабатывает влияния с типами: `users/login` (получает в свойстве `payload` простой объект с адресом электронной почты и паролем и выполняет вход в приложение) и `users/logout` (выполняет выход). Посредник `todosMiddleware()` обрабатывает влияние `todos/loadTodos`, которое загружает с бэкенда список дел.

Листинг 14.3. Посредники, обрабатывающие вход и выход посетителя и загрузку данных с бэкенда

```
import { login, logout, getTodos } from './api.js';

export function userMiddleware(store) {
    return function (next) {
        return async function (action) {
```

```
            switch (action.type) {
                case 'users/login': // ..................................  1
                    const user = await login(action.payload); // ..........  2
                    store.dispatch({ // ..................................  3
                        type: 'users/setUser',
                        payload: user
                    });
                    store.dispatch({ // ..................................  4
                        type: 'todos/loadTodos'
                    });
                    return;
                case 'users/logout': // ..................................  5
                    await logout(); // ...................................  6
                    store.dispatch({ // ..................................  7
                        type: 'users/setUser',
                        payload: null
                    });
                    store.dispatch({ // ..................................  8
                        type: 'todos/setTodos',
                        payload: []
                    });
                    return;
                default:
                    return next(action);
            }
        }
    }
}

export const todosMiddleware = store => next => async action => {
    if (action.type === 'todos/loadTodos') { // ............................  9
        const user = store.getState().currentUser; // ..................... 10
        const todos = await getTodos(user); // ............................ 11
        store.dispatch({ // ............................................... 12
            type: 'todos/setTodos',
            payload: todos
        });
    } else
        return next(action);
}
```

В посреднике `userMiddleware()`, если тип полученного влияния — `users/login` (поз. 1 в листинге 14.3), производим вход в приложение (поз. 2). Далее отправляем хранилищу влияние типа `users/setUser`, передав в его свойстве `payload` полученный объект со сведениями о посетителе, чтобы записать их в хранилище для дальнейшего использования (поз. 3). Влияние этого типа пройдет через оба посредника и

будет обработано преобразователем (см. листинг 14.2). Наконец, отправляем влияние `todos/loadTodos`, чтобы загрузить список дел (поз. 4, это влияние обрабатывается посредником `todosMiddleware()`, который мы рассмотрим позже).

В том же посреднике, в случае получения влияния `users/logout` (поз. 5), выполняем выход из приложения (поз. 6) и отправляем два влияния: `users/setUser` со значением `null` — чтобы стереть сведения о текущем посетителе (поз. 7), и `todos/setTodos` с пустым массивом — чтобы удалить загруженный ранее список дел (поз. 8).

В посреднике `todosMiddleware()`, если тип влияния — `todos/loadTodos` (поз. 9), сначала получаем объект со сведениями о текущем посетителе (поз. 10). Для простоты извлекаем его непосредственно из хранилища, не пользуясь селектором (что, вообще-то, не приветствуется). Далее загружаем с бэкенда список дел, созданных этим посетителем (поз. 11), и отправляем влияние `todos/setTodos` (поз. 12), которое запишет этот список дел в хранилище.

Осталось создать хранилище, привязав к нему только что созданных посредников:

```
import { createStore, applyMiddleware } from 'redux';
import mainReducer from './reducers.js';
import { userMiddleware, todosMiddleware } from './middleware.js';

const initialData = {
    todos: [],
    currentUser: null
};

const store = createStore(
    mainReducer,
    initialData,
    applyMiddleware(userChangedMiddleware, loadTodosMiddleware)
);
export default store;
```

Примеры использования созданного хранилища:

```
// Модуль src\Login.js
import store from './store.js';
export default function Login() {
    . . .
    const handleFormSubmit = evt => {
        evt.preventDefault();
        store.dispatch({
            type: 'users/login',
            payload: { email, password }
        });
    };
    . . .
}
```

```
// Модуль src\Logout.js
import { useEffect } from 'react';
import store from './store.js';
export default function Logout() {
    useEffect(() => {
        store.dispatch({ type: 'users/logout' });
    });
    . . .
}
```

14.2.2. Использование переходников. Библиотека Redux Thunk

Есть возможность немного сократить код посредников Redux, превратив их в переходники. *Переходник*[1] — это аналог посредника, но организованный несколько по-иному, проще и компактнее.

Чтобы хранилище Redux получило возможность обрабатывать переходники, к нему следует привязать особый посредник. Он входит в состав дополнительной библиотеки Redux Thunk, которую необходимо установить, набрав команду:

```
npm install redux-thunk
```

> ***На заметку***
>
> Полное описание библиотеки Redux Thunk приведено по интернет-адресу: **https://github.com/reduxjs/redux-thunk**.

Посредник, о котором идет речь, называется `thunk()` и объявлен в модуле `redux-thunk`.

> ***На заметку***
>
> В предыдущих версиях библиотеки этот посредник носил имя `thunkMiddleware()`.

Посредник следует привязать к хранилищу при его создании посредством функции `applyMiddleware()` (была описана в *разд. 14.2.1*). Пример:

```
import { createStore, applyMiddleware } from 'redux';
import { thunk } from 'redux-thunk';
. . .
const store = createStore(mainReducer, initialData, applyMiddleware(thunk));
```

Переходник представляет собой функцию, принимающую два параметра: ссылки на методы `dispatch()` и `getState()` хранилища. Никакого результата переходник возвращать не должен.

Выполнить переходник можно, просто передав ссылку на него методу `dispatch()` хранилища.

---

[1] В оригинальной документации — thunk.

Вот пример переходника `loadTodosThunk()`, загружающего список дел с бэкенда и аналогичного посреднику `todosMiddleware()` из листинга 14.3:

```
import { getTodos } from './api.js';
async function loadTodosThunk(dispatch, getState) {
    const user = getState().currentUser;
    const todos = await getTodos(user);
    dispatch({
        type: 'todos/setTodos',
        payload: todos
    });
}
```

А вот пример его использования:

```
store.dispatch(loadTodosThunk);
```

К сожалению, переходники не могут принимать дополнительные параметры. Однако можно объявить функцию, которая принимает необходимые параметры и возвращает переходник, в теле которого используются значения полученных параметров.

Вот пример функции `setDoneTodoThunkWrapper()`, которая возвращает переходник, помечающий дело с заданным идентификатором как выполненное и повторно загружающий список дел с бэкенда вызовом написанного ранее переходника `loadTodosThunk()`:

```
import { setDoneTodoOnBackend } from './api.js';
async function setDoneTodoThunkWrapper(key) {
    return (dispatch, getState) => {
        const user = getState().currentUser;
        await setDoneTodoOnBackend(user, key);
        dispatch(loadTodosThunk);
    };
}
```

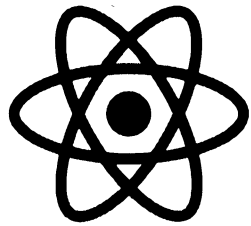
Пример вызова этого переходника:

```
store.dispatch(setDoneTodoThunkWrapper(item.key));
```

Посредник `thunk()` может быть использован совместно с любыми другими посредниками:

```
import { thunk } from 'redux-thunk';
import { userMiddleware } from './middleware.js';
. . .
const store = createStore(mainReducer, initialData,
                                       applyMiddleware(userMiddleware, thunk));
```

ГЛАВА 15



Библиотека Redux: централизованное хранение состояния, раздел 2

Применение централизованного хранилища состояния, реализуемого библиотекой Redux, позволяет существенно упростить обработку данных и сделать архитектуру веб-приложения более стройной и понятной.

Однако при программировании хранилища приходится писать довольно большой код. Несколько упростить работу помогут две дополнительные библиотеки, рассматриваемые далее.

15.1. Связывание React-компонентов с хранилищем Redux. Библиотека React Redux

Связанный компонент, в отличие от обычного, может работать с хранилищем Redux напрямую, получая данные, записанные в хранилище, и управляющие ими программные инструменты из пропов. Кроме того, связанный компонент подписан на изменения состояния хранилища и при каждом изменении находящихся там данных автоматически обновляется.

Дополнительная библиотека React Redux служит для создания связанных компонентов. Она устанавливается набором команды:

```
npm install react-redux
```

> ***НА ЗАМЕТКУ***
>
> Полное описание библиотеки React Redux находится по интернет-адресу: **https://react-redux.js.org/**.

15.1.1. Предоставление хранилища компонентам. Провайдер хранилища

Чтобы связанный компонент смог получить доступ к хранилищу, оно должно быть предоставлено ему особым провайдером.

Провайдер хранилища — это компонент, предоставляющий связанным компонентам доступ к хранилищу Redux (как провайдер контекста, описанный в *разд. 7.1.1.2*, предоставляет доступ к контексту).

Компонент `Provider`, реализующий провайдер хранилища, объявлен в модуле `react-redux`. Он поддерживает следующие пропы:

- `store` — хранилище, доступ к которому следует предоставить потомкам текущего провайдера;
- `context` — контекст, через который будет предоставляться хранилище (о контекстах рассказывалось в *разд. 7.1.1.2*). Если проп не указан, будет использоваться контекст по умолчанию, созданный самим провайдером.

  Предоставлять хранилище потомкам через контекст, отличный от используемого по умолчанию, может понадобиться в случае программирования сложных веб-приложений, в которых используется несколько хранилищ. В таком случае каждое хранилище следует предоставлять через отдельный, явно указанный контекст, чтобы оно не мешало другим хранилищам.

Связанные компоненты, которым нужно иметь доступ к указанному в текущем провайдере хранилищу, должны быть потомками этого провайдера, непосредственными или опосредованными.

Как правило, провайдер хранилища указывается в стартовом модуле, а его содержимым является либо компонент приложения, либо провайдер маршрутизации (см. *разд. 11.2.5*).

Пример:

```
import { Provider } from 'react-redux';
import store from './store.js';

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(
    <React.StrictMode>
        <Provider store={store}>
            <RouterProvider router={router} />
        </Provider>
    </React.StrictMode>
);
```

15.1.2. Доступ к хранилищу посредством функции *connect()*

Превратить компонент в связанный можно, применив функцию `connect()` из модуля `react-redux`:

```
connect([<получатель данных>=undefined[, <брокер влияний>=undefined[,
        <составитель пропов>=undefined[, <настройки>=undefined]]]])
```

Она возвращает в качестве результата другую, преобразующую функцию, являющуюся компонентом высшего порядка (см. *главу 10*). Она-то и выполняет преобразование обычного компонента, переданного ей в качестве единственного параметра, в связанный. Пример:

```
import { connect } from 'react-redux';

function __SomeComponent( . . . ) { . . . }

const connectSomeComponent = connect( . . . );
const SomeComponent = connectSomeComponent(___SomeComponent);
```

Параметры, принимаемые функцией `connect()`, будут подробно описаны далее, в отдельных разделах.

15.1.2.1. Функция *connect()*, параметр № 1: получатель данных

Получатель данных, указываемый первым параметром в вызове функции `connect()`, выполняет две задачи:

- подписывает связанный компонент на изменения состояния хранилища.

  В результате этого связанный компонент станет автоматически обновляться после каждого изменения состояния хранилища;
- указывает, какие данные, извлеченные из хранилища, и через какие пропы будет получать связанный компонент.

Получатель данных должен представлять собой функцию, принимающую один или два параметра:

- текущее состояние хранилища — обязательный. Обычно, по текущим соглашениям, ему дают имя `state`;
- объект с пропами, полученными связанным компонентом, — необязательный, обычно называемый `ownProps`. Если связанный компонент не принимает пропов, параметр не указывается.

Получатель данных будет вызываться:

- если он принимает один параметр — после каждого изменения состояния хранилища;
- если он принимает два параметра — после каждого изменения состояния хранилища или после изменения значения любого из пропов, полученных компонентом.

Получатель данных должен возвращать простой объект, свойства которого хранят требующиеся компоненту значения, полученные из хранилища. Имена этих свойств можно выбрать произвольно.

Свойства объекта, возвращенного получателем данных, вместе с хранящимися в них значениями будут переданы связанному компоненту в виде пропов.

Если в вызове функции `connect()` получатель данных не указан (или вместо него поставлено значение `undefined` или `null`), связанный компонент не будет ни полу-

чать с пропами данные из хранилища, ни обновляться при изменениях его состояния.

Пример получателя данных, извлекающего из хранилища список дел, передающего его связанному компоненту в пропе `list` и вызываемого только при изменении состояния хранилища:

```
const listReceiver = state => ({ list: state.todos });
```

Пример использования этого получателя данных для создания связанного компонента `TodoList`, получающего список дел из хранилища:

```
import { connect } from 'react-redux';

function __TodoListUnconnected(props) { // ..............................  1
    . . .
    return (
        <table>
            {props.list.map(item => ( // ..................................  2
                . . .
            ))}
        </table>
    );
}

const connectTodoList = connect(listReceiver); // ........................  3
const TodoList = connectTodoList(__TodoList); // .........................  4
export default TodoList;
```

Уже при написании изначального, несвязанного компонента (поз. 1 в приведенном примере) учитываем, что он будет получать список дел через проп `list` (поз. 2).

Вызываем функцию `connect()`, передав ей написанный ранее получатель данных, и получаем на выходе преобразующую функцию (поз. 3). Вызываем преобразующую функцию, передав ей изначальный компонент, и получаем связанный компонент, пригодный к использованию в проекте (поз. 4).

Вот пример получателя данных, извлекающего дело с идентификатором, переданным через проп `key`, и вызываемого как после изменения состояния хранилища, так и после изменения значения любого пропа:

```
const itemReceiver = (state, ownProps) => ({
    item: state.todos.find(current => current.key === ownProps.key);
});
```

15.1.2.2. Функция *connect()*, параметр № 2: брокер влияний

Брокер влияний, указываемый вторым параметром в вызове функции `connect()`, предоставляет связанному компоненту набор функций, посредством которых тот сможет отправлять хранилищу влияния и тем самым управлять находящимися там данными. Эти функции передаются компоненту также через пропы.

Брокер влияний должен представлять собой функцию, принимающую один или два параметра:

- метод `dispatch()` хранилища — обязательный, называемый, как правило, `dispatch`;
- объект с пропами, полученными связанным компонентом, — необязательный, обычно называемый `ownProps`.

Если брокер влияний принимает один параметр, он будет вызван единственный раз — при преобразовании компонента в связанный. Если же брокер принимает два параметра, он также будет вызываться после изменения значения любого из пропов, полученных связанным компонентом.

Брокер влияний должен возвращать простой объект, свойства которого хранят требуемые компоненту функции, манипулирующие данными в хранилище. Эти функции могут принимать произвольные параметры, отправлять влияния непосредственно, использовать создатели (см. *разд. 14.1.8*) и отправители (см. *разд. 14.1.9*) влияний, вызывать переходники (см. *разд. 14.2.2*). Свойствам, хранящим эти функции, можно давать произвольные имена.

Свойства объекта, возвращенного брокером влияний, вместе с записанными в них функциями будут переданы связанному компоненту в виде пропов.

Если в вызове функции `connect()` брокер влияний не указан (или вместо него поставлено значение `undefined` или `null`), связанный компонент получит дополнительный проп `dispatch`, хранящий метод `dispatch()` хранилища. Так что связанный компонент даже в этом случае сможет управлять данными, находящимися в хранилище.

Пример брокера влияний, выдающего объект с двумя функциями: первая — помечает указанное дело как выполненное, а вторая — удаляет указанное дело:

```
const todoActionBroker = dispatch => ({
    setDone: key => {dispatch({ type: 'todos/setDone', payload: key })},
    del: key => {dispatch({ type: 'todos/delete', payload: key })}
});
```

Пример использования этого брокера влияний (совместно с получателем данных из *разд. 15.1.2.1*):

```
function __TodoList(props) {
    return (
        <table>
            {props.list.map(item => (
                <tr key={item.key}>
                    . . .
                    <button title="Пометить как выполненное"
                            onClick={() => {props.setDone(item.key)}}
                    >
                        &#9745;
                    </button>
                    . . .
```

```
                    <button title="Удалить"
                            onClick={() => {props.del(item.key)}}
                    >
                        &#9746;
                    </button>
                    . . .
                </tr>
            ))}
        </table>
    );
}

const connectTodoList = connect(listReceiver, todoActionBroker);
const TodoList = connectTodoList(__TodoList);
```

15.1.2.3. Функция *connect()*, параметр № 3: составитель пропов

Составитель пропов, задаваемый третьим параметром в вызове функции `connect()`, управляет составом пропов, передаваемых связанному компоненту.

Составитель пропов должен представлять собой функцию, принимающую три параметра:

- результат, возвращенный получателем данных (этому параметру традиционно дают имя `stateProps`);
- результат, возвращенный брокером влияний (с именем `dispatchProps`);
- простой объект с пропами, переданными компоненту в JSX-коде (с именем `ownProps`).

Составитель пропов должен возвращать простой объект, свойства которого вместе с хранящимися в нем значениями будут преобразованы в пропы и переданы связанному компоненту.

Если в вызове функции `connect()` составитель пропов не указан (или вместо него поставлено `undefined` или `null`), связанный компонент получит в качестве пропов все свойства из результатов, возвращенных получателем данных и брокером влияний, и все пропы, переданные компоненту в JSX-коде.

Далее приведен код составителя пропов, который сводит результаты, возвращенные получателем данных и брокером влияний, и пропы, переданные компоненту, в простые объекты и заносит их в результирующие пропы `storage`, `actions` и `own` соответственно:

```
const propMaker = (stateProps, dispatchProps, ownProps) => ({
    storage: stateProps,
    actions: dispatchProps,
    own: ownProps
});
```

Пример использования этого составителя пропов:

```
function __TodoList(props) {
    return (
        <table>
            {props.storage.list.map(item => (
                    . . .
                    <button title="Пометить как выполненное"
                            onClick={() => {props.actions.setDone(item.key)}}
                    >
                        &#9745;
                    </button>
                    . . .
            ))}
        </table>
    );
}

const connectTodoList = connect(listReceiver, todoActionBroker, propMaker);
const TodoList = connectTodoList(__TodoList);
```

15.1.2.4. Функция *connect()*, параметр № 4: настройки

Дополнительные *настройки* связанного компонента указываются в виде простого объекта со следующими свойствами:

- `areStatesEqual` — функция, сравнивающая предыдущее и текущее состояния хранилища.

  Она должна принимать четыре параметра:

  - предыдущее состояния хранилища;
  - текущее состояние хранилища;
  - предыдущие значения пропов, переданных компоненту в JSX-коде, в виде простого объекта;
  - текущие значения пропов.

  Последние два параметра следует указывать, если для проведения сравнения предыдущего и текущего состояний требуются значения пропов, полученных компонентом. Если пропы для сравнения не нужны, последние два параметра указывать необязательно.

  Функция должна возвращать `true`, если состояния равны (т. е. состояние хранилища не изменилось), или `false` — если не равны (состояние изменилось).

  Пример функции, сравнивающей только размеры предыдущего и текущего массивов дел (в результате компонент будет обновлен лишь в том случае, если размер массива дел изменился):

```
const options = {
    areStatesEqual: (prev, next) =>
                    prev.data.length === next.data.length
};
const connectSomeComponent = connect(. . ., options);
```

Пример функции, предписывающей обновить компонент, даже если состояние хранилища не изменилось:

```
const options = { areStatesEqual: () => false };
```

Она может пригодиться, если преобразователь или какой-либо из посредников написан некорректно и вместо замены всего состояния изменяет записанные в нем значения.

Если эта настройка не указана, будет выполняться строгое сравнение предыдущего состояния с текущим. Будучи записанным в виде выражения JavaScript, сказанное выглядит так:

```
const options = { areStatesEqual: (prev, next) => prev === next };
```

- `areOwnPropsEqual` — функция, сравнивающая предыдущие и текущие значения полученных компонентом пропов.

  Она должна принимать в качестве двух параметров простые объекты с предыдущими и текущими значениями пропов и возвращать `true`, если все сравниваемые пропы равны, или `false` — если хотя бы один из пропов изменился.

  Пример функции, предписывающей обновить компонент лишь при изменении значения пропа `key`:

```
const options = {
    areOwnPropsEqual: (prev, next) => prev.key === next.key
};
```

  Если эта настройка не указана, будет выполняться строгое сравнение значений *всех* полученных компонентом пропов;

- `areStatePropsEqual` — функция, которая сравнивает предыдущий и текущий результаты, возвращенные получателем данных (см. *разд. 15.1.2.1*).

  Она должна принимать в качестве двух параметров простые объекты с предыдущим и текущим результатами, выданными получателем данных, и возвращать `true`, если все сравниваемые значения равны, или `false` — если хотя бы одно из них изменилось.

  Эта настройка может пригодиться, если какое-либо из значений, выдаваемых получателем данных, не берется из хранилища напрямую, а вырабатывается в результате вычислений.

  Если эта настройка не указана, будет выполняться строгое сравнение *всех* значений, выданных получателем данных;

- `areMergedPropsEqual` — функция, которая сравнивает предыдущий и текущий результаты, возвращенные составителем пропов (см. *разд. 15.1.2.3*).

Она должна принимать в качестве двух параметров простые объекты с предыдущим и текущим результатами, выданными составителем пропов, и возвращать `true`, если все сравниваемые значения равны, или `false` — если хотя бы одно из них изменилось.

Эта настройка может пригодиться, если какое-либо из значений, выдаваемых составителем пропов, вырабатывается в результате вычислений непосредственно в составителе.

Если настройка не указана, будет выполняться строгое сравнение *всех* значений, выданных составителем пропов;

- `context` — контекст, через который предоставляется хранилище. Следует указать тот же контекст, что был задан в пропе `context` провайдера хранилища (см. *разд. 15.1.1*), в противном случае возникнет критическая ошибка. Если настройка не указана, будет использоваться контекст по умолчанию.

> ***НА ЗАМЕТКУ***
>
> Функция `connectAdvanced()`, аналогичная функции `connect()`, только имеющая другой формат вызова, в настоящее время объявлена устаревшей и не рекомендованной к применению.

15.1.3. Доступ к хранилищу посредством хуков

Компоненты могут получить доступ к хранилищу Redux также посредством хуков, предоставляемых библиотекой React Redux.

15.1.3.1. Получение данных из хранилища

Получить какое-либо значение из хранилища позволяет хук `useSelector()` из модуля `react-redux`. Кроме того, этот хук подписывает компонент на изменения состояния хранилища. Формат вызова хука:

```
useSelector(<получатель значения>[, <функция сравнения>=undefined])
```

Получатель значения должен представлять собой функцию, принимающую в качестве единственного параметра текущее состояние хранилища и возвращающую значение, которое требуется получить. Значение, возвращенное *получателем*, будет возвращено самим хуком.

Хук `useSelector()` вызывается повторно при любом изменении состояния хранилища. Если возвращенное им значение не совпадает с возвращенным при предыдущем вызове, компонент обновляется. Если же оба значения равны, компонент обновлен не будет.

Пример получения из хранилища массива дел:

```
import { useSelector } from 'react-redux';
export default function TodoList() {
    const list = useSelector(state => state.todos);
    . . .
}
```

Хук может быть вызван произвольное количество раз — для получения из хранилища разных значений:

```
const list = useSelector(state => state.todos);
const currentUser = useSelector(state => state.currentUser);
```

По умолчанию хук сравнивает текущее и предыдущее значения, выданные *получателем*, с помощью оператора строгого сравнения `===`. Однако можно указать свою *функцию сравнения*. Она должна принимать в качестве двух параметров текущее и предыдущее значения и возвращать `true`, если они равны, и `false` — в противном случае.

Пример получения массива дел, причем массивы считаются разными, если различаются их размеры:

```
const list = useSelector(state => state.todos,
           (currentList, prevList) => currentList.length === prevList.length);
```

В *получателе значения* можно использовать значения пропов, полученных компонентом:

```
export default function TodoDetail({ itemKey }) {
    const todo = useSelector(
        state => state.todos.find(current => current.key === itemKey)
    );
    . . .
}
```

Получатели значений для повышения производительности можно мемоизировать, применив библиотеку Reselect (см. *разд. 14.1.10.3*).

Пример мемоизации *получателя*, работающего только с состоянием хранилища:

```
import { createSelector } from 'reselect';

const listLengthReceiver = createSelector(
    state => state.todos,
    todos => todos.length
);

export default function TodoListLength() {
    const listLength = useSelector(state => listLengthReceiver(state));
    return <p>Всего {listLength} запланированных дел.</p>;
}
```

Пример мемоизации *получателя*, работающего также и с пропами компонента:

```
const itemReceiver = createSelector(
    state => state.todos,
    (state, itemKey) => itemKey,
    (todos, itemKey) => todos.find(current => current.key === itemKey)
);
```

```
export default function TodoItem({ itemKey }) {
    const item = useSelector(state => itemReceiver(state, itemKey));
    . . .
}
```

15.1.3.2. Управление данными в хранилище

Хук `useDispatch()` из модуля `react-redux` возвращает ссылку на метод `dispatch()` хранилища, что позволяет управлять находящимися в этом хранилище данными:

```
import { useDispatch } from 'react-redux';

export default function TodoList() {
    const dispatch = useDispatch();

    return (
        . . .
        <button title="Пометить как выполненное"
                onClick={() => dispatch(
                    { type: 'todos/setDone', payload: item.key }
                )}
        >
            &#9745;
        </button>
        . . .
    );
}
```

15.1.3.3. Получение доступа к самому хранилищу

Хук `useStore()` из модуля `react-redux` возвращает объект хранилища:

```
import { useStore } from 'react-redux';

export default function TodoList() {
    const store = useStore();
    const list = store.getState().todos;

    return (
        . . .
        <button title="Пометить как выполненное"
                onClick={() => store.dispatch(
                    { type: 'todos/setDone', payload: item.key }
                )}
        >
            &#9745;
        </button>
        . . .
    );
}
```

15.1.3.4. Работа с хранилищем, предоставляемым через пользовательский контекст

Провайдер хранилища позволяет передавать хранилище потомкам через произвольный контекст, указанный разработчиком (подробности — в *разд. 15.1.1*).

Хуки `useSelector()`, `useDispatch()` и `useStore()`, описанные в *разд. 15.1.3.1–15.1.3.3*, обеспечивают работу с хранилищем, предоставляемым через контекст по умолчанию. Если хранилище предоставляется через другой контекст, использовать эти хуки для доступа к нему нельзя.

Вместо этого следует воспользоваться следующими функциями, объявленными в модуле `react-redux`:

- `createSelectorHook(<контекст>)` — возвращает хук, служащий для получения данных из хранилища, которое предоставляется через указанный *контекст*, и аналогичный хуку `useSelector()`;
- `createDispatchHook(<контекст>)` — возвращает хук, служащий для управления данными в хранилище, которое предоставляется через указанный *контекст*, и аналогичный хуку `useDispatch()`;
- `createStoreHook(<контекст>)` — возвращает хук, служащий для получения доступа к хранилищу, которое предоставляется через указанный *контекст*, и аналогичный хуку `useStore()`.

Пример:

```
// Модуль src\index.js
import { createContext } from 'react';

export const storeContext = createContext(undefined);

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(
    <React.StrictMode>
        <Provider store={store}context={storeContext}>
            <RouterProvider router={router} />
        </Provider>
    </React.StrictMode>
);

// Модуль src\TodoList.js
import { createSelectorHook, createDispatchHook } from 'react-redux';
import { storeContext } from './index';

const contextUseSelector = createUseSelector(storeContext);
const contextUseDispatch = createUseDispatch(storeContext);

export default function TodoList() {
  const list = contextUseSelector((state) => state.data);
```

```
  const dispatch = contextUseDispatch();
  . . .
}
```

15.2. Упрощение создания хранилища. Библиотека Redux Toolkit

Дополнительная библиотека Redux Toolkit позволяет заметно упростить и сделать более понятным код, создающий хранилище Redux. Она устанавливается командой:

```
npm install @reduxjs/toolkit
```

> ***На заметку***
>
> Полное описание библиотеки Redux Toolkit приведено по интернет-адресу: **https://redux-toolkit.js.org/**.

15.2.1. Упрощение формирования создателей влияний

Упростить формирование создателей влияний (описаны в *разд. 14.1.8*) призвана функция `createAction()` из модуля `@reduxjs/toolkit`:

```
createAction(<тип влияния>[, <подготовитель значения>=undefined])
```

Тип влияния указывается в виде строки.

Функция `createAction()` возвращает другую функцию — сформированный создатель влияния. По умолчанию он принимает единственный параметр, значение которого самостоятельно занесет в свойство `payload` создаваемого влияния.

Пример формирования создателя влияний, заносящих в состояние хранилища заданный массив дел:

```
import { createAction } from '@reduxjs/toolkit';
export const createSetTodosAction = createAction('todos/setTodos');
```

Пример использования этого создателя влияния в коде посредника:

```
import { getTodos } from './api.js';
import { createSetTodosAction } from './actionCreators.js';
. . .
const user = store.getState().currentUser;
const todos = await getTodos(user);
store.dispatch(createSetTodosAction(todos));
```

Недостатком такого рода создателей влияний является то, что они принимают лишь один параметр и заносят его значение непосредственно в свойство `payload` создаваемого влияния. Если создателю требуется передать несколько значений или занести в свойство `payload` какую-либо величину, полученную на основе обработки заданного значения (значений), следует избрать другой подход.

В вызове функции `createAction()` можно указать *подготовитель значения*. Он должен представлять собой функцию, принимающую произвольное количество параметров (как минимум один). С ними *подготовителю* будут передаваться значения всех параметров, переданных при вызове создателю влияний. *Подготовитель* должен вернуть в качестве результата простой объект со свойством `payload`, хранящим вычисленное значение. На основе возвращенного им объекта и будет создано влияние.

Вот пример формирования создателя влияний, помечающих дело как выполненное, который принимает в качестве параметра объект дела и заносит его идентификатор в свойство `payload` создаваемого влияния:

```
export const createSetDoneItemAction = createAction('todos/setDoneItem',
    item => ({ payload: item.key })
);
```

Пример его использования:

```
import { createSetDoneItemAction } from './actionCreators.js';
. . .
<button title="Пометить как выполненное"
        onClick={() => {store.dispatch(createSetDoneItemAction(item))}}>
    &#9745;
</button>
```

Каждый создатель влияний, созданный функцией `createAction()`, поддерживает метод `match(<влияние>)`. Он возвращает `true`, если заданное *влияние* относится к тому же типу, что был указан при формировании текущего создателя влияний, и `false` — в противном случае. Пример:

```
if (createSetTodosAction.match(action))
    // Влияние action относится к типу todos/setTodos
```

15.2.2. Упрощение создания преобразователя

Существенно упростить программирование единственного преобразователя в приложении позволяет функция `createReducer()` из модуля `@reduxjs/toolkit`:

```
createReducer(<изначальное состояние>, <строящая функция>)
```

Изначальное состояние будет занесено в хранилище при первом вызове создаваемого преобразователя. *Строящая функция* служит для формирования структуры создаваемого преобразователя и будет рассмотрена чуть позже.

Функция `createReducer()` вернет преобразователь, созданный на основе указанных параметров, который можно передать функции `createStore()`, создающей хранилище:

```
import { createReducer } from '@reduxjs/toolkit';
. . .
const reducer = createReducer(initialData, . . . );
const store = createStore(reducer);
```

Вернемся к *строящей функции*, указываемой в вызове функции `createReducer()`. Она должна принимать единственным параметром объект *построителя*, на основе которого и будет формироваться структура создаваемого преобразователя.

Построитель поддерживает три метода, которые конструируют логику работы формируемого преобразователя. Каждый метод добавляет в преобразователь сущность, хранящую заданные *условие* и *функцию*. В результате все поступающие влияния, удовлетворяющие заданному *условию*, будут обрабатываться указанной *функцией*.

Все методы построителя в качестве результата возвращают объект текущего построителя, что позволяет записывать их вызовы цепочкой.

Вот методы, поддерживаемые построителем:

- `addCase(<тип влияния>, <обработчик>)` — предписывает обрабатывать все влияния, относящиеся к заданному *типу*, указанной функцией-*обработчиком* (создавая тем самым *строгое условие*). Вызов этого метода аналогичен добавлению конструкции `case` в выражение выбора JavaScript.

  Тип влияния может быть указан как в виде строки, так и в виде создателя влияний, сформированного вызовом функции `createAction()` (см. *разд. 15.2.1*).

  Функция-*обработчик* должна принимать два параметра. Первым передается прокси-объект, перехватывающий все операции с состоянием хранилища и производящий эти операции с копией состояния. Вторым параметром передается очередное поступившее влияние.

  Обработчик должен внести соответствующие изменения в состояние, представляемое прокси-объектом, который был получен с первым параметром. Результата он возвращать не должен.

  Пример:

```
import { createReducer } from '@reduxjs/toolkit';
import { createSetTodosAction } from './actionCreators.js';

const reducer = createReducer(initialData, builder => {
    builder
        .addCase(createSetTodosAction, (state, action) => {
            state.todos = action.payload;
        })
        .addCase('todos/add', (state, action) => {
            state.todos.push(action.payload);
        })
        .addCase('todos/setDone', (state, action) => {
            const deed = state.todos.find(
                current => current.key === action.payload
            );
            if (deed)
                deed.done = true;
        })
```

```
        .addCase('todos/delete', (state, action) => {
            state.todos.splice(action.payload, 1);
        })
        .addCase('users/setUser', (state, action) => {
            state.currentUser = action.payload;
        });
});
```

Впрочем, если требуется радикально изменить состояние хранилища, из *обработчика* можно вернуть полностью новое состояние, например:

```
.addCase('todos/setTodos', (state, { payload }) =>
                              ({ ...state, todos: payload }))
```

Из всех строгих условий, созданных в преобразователе вызовами метода `addCase()`, будет исполнено либо *лишь одно* — то, чей тип совпадает с типом поступившего влияния, либо *ни одного* — если же тип очередного влияния не совпадет ни с одним из указанных в имеющихся строгих условиях;

- `addMatcher(<`*функция проверки*`>, <`*обработчик*`>)` — предписывает обрабатывать все влияния, в отношении которых заданная *функция проверки* вернет `true`, указанной функцией-*обработчиком* (формируя тем самым *нестрогое условие*).

  Функция проверки должна принимать единственным параметром очередное поступившее влияние, реализовывать его проверку на соответствие заданным условиям и возвращать `true`, если влияние удовлетворяет этим условиям, и `false` — в противном случае.

  Обработчик должен соответствовать тем же требованиям, что и у метода `addCase()`.

  > ***Внимание!***
  >
  > Вызовы метода `addMatcher()`, создающие нестрогие условия, должны следовать за вызовами метода `addCase()`, которые задают строгие условия.

Пример:

```
function isPayloadTooBig(action) {
    return action.payload > 100;
}
function isPayloadTooSmall(action) {
    return action.payload <= 100;
}
function isPayloadEven(action) {
    return action.payload % 2 === 0;
}
function isPayloadOdd(action) {
    return action.payload % 2 !== 0;
}
```

```
const reducer = createReducer(initialData, builder => {
    builder
        .addCase('counter/increment', (state, { payload=1 }) => { // ....  1
            state.count += payload;
        })
        .addCase('counter/decrement', (state, { payload=1 }) => { // ....  2
            state.count -= payload;
        })
        .addMatcher(isPayloadTooBig, (state, action) => { // ............  3
            state.tooBig = true;
        })
        .addMatcher(isPayloadTooSmall, (state, action) => { // ..........  4
            state.tooBig = false;
        })
        .addMatcher(isPayloadEven, (state, action) => { // ..............  5
            state.even = true;
        })
        .addMatcher(isPayloadOdd, (state, action) => { // ...............  6
            state.even = false;
        });
});
```

Будут выполнены *все* нестрогие условия, созданные в преобразователе, у которых функции проверки вернут `true`, или *ни одно* из них, если у всех этих условий функции проверки вернут `false`. Нестрогие условия проверяются и выполняются в том порядке, в котором описаны в коде строящей функции.

Например, при поступлении в преобразователь, код которого был приведен только что, влияния с типом `counter/increment` и значением свойства `payload`, равным 50, будут выполнены первое строгое условие (поз. 1), а также второе (поз. 4) и третье (поз. 5) нестрогие условия. А при поступлении влияния `counter/decrement` со значением 409 будут выполнены второе строгое условие (поз. 2), а также первое (поз. 3) и четвертое (поз. 6) нестрогие условия;

- `addDefaultCase(<обработчик>)` — задает функцию-*обработчик*, призванную обрабатывать влияния, с которыми не совпало ни одно из заданных строгих и нестрогих условий (задавая *обработчик по умолчанию*).

  Обработчик должен принимать единственный параметр — прокси-объект, формирующий копию состояния хранилища.

> ***Внимание!***
>
> **Вызов метода `addDefaultCase()` должен следовать за вызовами методов `addCase()` и `addMatcher()`. Допускается лишь один вызов метода `addDefaultCase()` в теле строящей функции.**

Пример:

```
const reducer = createReducer(initialData, builder => {
    builder
        . . .
```

```
        .addDefaultCase(state => { state.count += 1; })
});
```

На заметку

Альтернативный формат вызова функции `createReducer()`, **в котором функции передаются четыре параметра, больше не поддерживается.**

15.2.3. Упрощение создания хранилища

Немного упростить создание хранилища Redux позволяет функция `configureStore(<настройки>)` из модуля `@reduxjs/toolkit`. *Настройки* указываются в виде простого объекта со следующими свойствами:

- `reducer` — либо единственный преобразователь, либо набор отдельных преобразователей (указание набора преобразователей рассмотрено в *разд. 15.2.4*):

  ```
  import { configureStore } from '@reduxjs/toolkit';
  . . .
  const store = configureStore({ reducer: rootReducer });
  ```

  Это единственная обязательная для указания настройка. Остальные можно не указывать;

- `middleware` — функция, возвращающая массив посредников, которые следует привязать к создаваемому хранилищу.

  Функция должна принимать в качестве единственного параметра ссылку на функцию `getDefaultMiddleware()`, которая возвращает массив посредников, привязываемых по умолчанию. Необходимые посредники должны быть добавлены в этот массив.

  Пример:

  ```
  const store = configureStore({
      reducer: rootReducer,
      middleware: getDefaultMiddleware =>
          getDefaultMiddleware().concat(
              userChangedMiddleware,
              loadDataMiddleware,
              addItemMiddleware,
              setDoneItemMiddleware,
              deleteItemMiddleware
          )
  });
  ```

  В состав привязываемых по умолчанию к хранилищу входят следующие посредники:

  - Redux Thunk (описан в *разд. 14.2.2*) — и при разработке, и при эксплуатации.

    Следующие посредники привязываются только в том случае, если проект запущен в режиме разработки:

- посредник, выявляющий прямое изменение состояния хранилища (вместо создания его копии) и возбуждающий в таком случае исключение;
- посредник, выявляющий передачу в составе влияний данных недопустимых типов: функций, промисов, значений типа `Symbol` и некоторых других. При попытке передать значение недопустимого типа возбуждается исключение;
- посредник, выявляющий ошибку при отправке создателя влияния вида:

  ```
  store.dispatch(createSetTodosAction);
  ```

  Правильный код:

  ```
  store.dispatch(createSetTodosAction(todos));
  ```

  В случае выявления такой ошибки также возбуждается исключение.

Если посредники, привязываемые по умолчанию, не требуются, задаваемая функция может просто вернуть массив требуемых посредников:

```
const store = configureStore({
    reducer: rootReducer,
    middleware: () => [
        userChangedMiddleware,
        loadDataMiddleware,
        addItemMiddleware,
        setDoneItemMiddleware,
        deleteItemMiddleware
    ]
});
```

Если настройка не указана, к хранилищу будут привязаны посредники, используемые по умолчанию (возвращенные неявно вызванной функцией `getDefaultMiddleware()`);

- `preloadedState` — изначальное состояние хранилища (если оно не было указано при объявлении преобразователя);
- `devTools` — `true`, чтобы в число расширителей хранилища были добавлены средства для взаимодействия с расширением веб-обозревателя Redux DevTools, применяемом при отладке (подробности приведены в *главе 23*), и `false`, чтобы не добавлять этот расширитель (по умолчанию: `true`);
- `enhancers` — функция, возвращающая массив расширителей, которые следует привязать к создаваемому хранилищу.

  Функция должна принимать в качестве единственного параметра ссылку на функцию `getDefaultEnhancers()`, которая возвращает массив расширителей, привязываемых по умолчанию. Необходимые расширители должны быть добавлены в этот массив.

  Пример:

  ```
  const store = configureStore({
      . . .
  ```

```
    enhancers: getDefaultEnhancers =>
        getDefaultEnhancers().concat(
            throttleEnhancer,
            offlineEnhancer
        )
});
```

По умолчанию к хранилищу будут привязаны следующие расширители:

- `applyMiddleware()` (см. *разд. 14.2.1*);
- `devToolsExtension()` — средства для взаимодействия с расширением веб-обозревателя Redux DevTools. Добавляется, если у параметра `devTools` указано значение `true`.

Если расширители, привязываемые по умолчанию, не нужны, задаваемая функция может просто вернуть массив необходимых расширителей:

```
const store = configureStore({
    . . .
    enhancers: () => [
        throttleEnhancer,
        offlineEnhancer
    ]
});
```

Если настройка не указана, к хранилищу будут привязаны расширители, используемые по умолчанию (возвращенные неявно вызванной функцией `getDefaultEnhancers()`).

> ***НА ЗАМЕТКУ***
>
> Указание привязываемых посредников и расширителей в настройках `middleware` и `enhancers` в виде массивов больше не поддерживается.

15.2.4. Упрощение создания преобразователей срезов

Redux Toolkit всячески подталкивает разработчиков к написанию преобразователей, манипулирующих отдельными срезами хранилища (см. *разд. 14.1.10.1*), поскольку в этом случае программирование упрощается весьма существенно.

Для создания преобразователя среза применяется функция `createSlice(<настройки>)` из модуля `@reduxjs/toolkit`. В качестве *настроек* указывается простой объект со следующими свойствами:

- `name` — имя среза, используемое для формирования типов влияний (подробности — чуть позже).

  Нужно отметить, что имя свойства состояния хранилища, в котором будут храниться данные среза, задается отдельно и может не совпадать с указанным здесь именем среза;

- `reducers` — набор обработчиков влияний, которые должны содержаться в создаваемом преобразователе среза, в виде простого объекта.

Свойства этого объекта зададут суффиксы типов влияний, обрабатываемых преобразователем (подробности — чуть позже). В качестве значений этих свойств можно указать одно из двух:

- функцию-обработчик.

  Первым параметром эта функция получит прокси-объект, перехватывающий все операции с данными среза и производящий эти операции с копией указанных данных. Вторым параметром передается поступившее влияние.

  Обработчик должен внести соответствующие изменения в данные среза, представляемые прокси-объектом из первого параметра. Результата он возвращать не должен.

  Однако, если требуется внести в данные среза значительные правки или заменить их полностью, следует вернуть из обработчика *новые данные среза* (не новое состояние хранилища!);

- простой объект со свойствами:
  - `reducer` — функция-обработчик;
  - `prepare` — функция-подготовитель значения (см. *разд. 15.2.1*).

На основе имен свойств объекта, заданного в настройке `reducers`, будут сгенерированы типы влияний, обрабатываемых преобразователем. Генерирование этих типов выполняется согласно шаблону:

<имя среза>/<имя свойства объекта>

Имя среза берется из настройки `name`;

- `selectors` — набор селекторов, извлекающих значения из среза, в виде простого объекта.

  Имена свойств этого объекта зададут имена создаваемых селекторов. Значениями этих свойств должны быть функции, возвращающие требуемые значения из хранилища. Каждая из этих функций первым параметром должна принимать текущее состояние среза, а следующими параметрами — значения всех параметров, которые были переданы соответствующему селектору при вызове;

- `initialState` — изначальное состояние среза;

- `extraReducers` — дополнительный преобразователь, содержащий обработчики влияний с произвольно заданными типами. Указывается в виде строящей функции (см. *разд. 15.2.2.1*).

> ***На заметку***
>
> Задание значения настройки `extraReducers` в виде простого объекта, аналогичного указываемому в настройке `redurecs`, больше не поддерживается.

Функция `createSlice()` возвращает объект, описывающий созданный преобразователь среза и содержащий следующие свойства:

- `reducer` — сам созданный преобразователь среза;
- `actions` — набор готовых создателей влияний в виде простого объекта;
- `selectors` — набор готовых селекторов в виде простого объекта.

В листинге 15.1 представлен код двух преобразователей срезов: первый — обрабатывает массив дел, второй — сведения о текущем посетителе.

Листинг 15.1. Преобразователи срезов

```
import { createSlice } from '@reduxjs/toolkit';

export const todosReducer = createSlice({
    name: 'todos',
    initialState: [],
    reducers: {
        setTodos: (state, { payload }) => payload, // ..................... 2
        add: (state, { payload }) => { // ................................. 1
            state.push(payload);
        },
        setDone: (state, { payload }) => {
            const deed = state.find(
                current => current.key === payload
            );
            if (deed)
              deed.done = true;
        },
        del: (state, { payload }) => {
            state.splice(payload, 1);
        }
    },
    selectors: {
        getTodos: state => state,
        getTodo: (state, key) => state.find(current => current.key === key)
    }
});

export const userReducer = createSlice({
    name: 'user',
    initialState: null,
    reducers: {
        setUser: (state, { payload }) => payload // ....................... 3
    },
    selectors: {
        getUser: state => state;
    }
});
```

Незначительные изменения в данные среза можно внести непосредственно, манипулируя прокси-объектом, полученным с первым параметром (поз. 1 в листинге 15.1). Если же требуется полностью заменить существующие данные среза новыми, следует вернуть из обработчика новые данные (поз. 2 и 3).

При создании хранилища на основе нескольких преобразователей среза нужно передать функции `configureStore()` (см. *разд. 15.2.3*) набор преобразователей. Он должен представлять собой простой объект, свойства которого хранят отдельные преобразователи срезов. Свойства набора зададут срезы хранилища, в которых будут храниться данные, обрабатываемые соответствующими преобразователями. Этим свойствам можно дать произвольные имена.

Пример:

```
import { todosReducer, userReducer } from './reducers.js';
. . .
const store = configureStore({
    reducer: {
        todos: todosReducer.reducer,
        currentUser: userReducer.reducer
    },
    . . .
});
```

Отправлять влияния созданному таким образом хранилищу можно способами, описанными в *главе 14*, используя типы, сгенерированные автоматически при создании преобразователей среза (принципы формирования типов были описаны ранее):

```
store.dispatch({
    type: 'user/setUser',
    payload: loggedInUser
});
. . .
store.dispatch({
    type: 'todos/add',
    payload: addedDeed
});
```

Однако проще прибегнуть к создателям влияний, сгенерированным автоматически при создании преобразователей:

```
const { setTodos, add, setDone, del } = todosReducer.actions;
. . .
store.dispatch(setTodos(todos));
```

Для доступа к данным, хранящимся в срезе, удобно пользоваться селекторами, также сгенерированными автоматически:

```
const { getTodos, getTodo } = todosReducer.actions;
. . .
const todos = getTodos();
const todo = getTodo(key);
```

15.2.4.1. Допустимые и недопустимые типы данных во влияниях

Если срез хранилища хранит какой-либо объект, обязательно следует удостовериться, что свойства этого объекта содержат лишь значения допустимых библиотекой Redux Toolkit типов. То же самое касается и значений, пересылаемых в составе влияний.

Допустимыми типами являются строки, числа, логические величины, массивы, простые объекты и `null`. Все остальные типы являются *недопустимыми*.

Встретив значение недопустимого типа в хранилище или во влиянии, Redux Toolkit выведет в консоли веб-обозревателя сообщение о некритической ошибке. А в ряде случаев наличие такого значения вызовет возникновение критической ошибки в программном ядре библиотеки. Такие ошибки крайне трудно локализовать, поэтому лучше избегать их возникновения.

В качестве примера можно привести объект со сведениями о текущем посетителе, выдаваемый библиотекой Google Firebase SDK и передаваемый с влиянием типа `user/setUser`. Этот объект содержит свойства, хранящие недопустимые с точки зрения Redux Toolkit значения, которые способны привести к краху приложения. Поэтому с влиянием следует передавать простой объект, содержащий лишь те сведения о посетителе, которые нужны для работы и относятся к допустимым типам. Пример:

```
store.dispatch({
    type: 'user/setUser',
    payload: user ? { uid: user.uid, email: user.email } : null
});
```

15.2.5. Упрощение создания переходников

Redux Toolkit позволяет значительно упростить разработку переходников и сделать их код намного понятнее. Это достигается за счет отделения кода, запускающего выполнение асинхронного действия в переходнике, от кода, который обрабатывает результат его завершения — успешного или неуспешного.

15.2.5.1. Создание переходников

Для создания переходника применяется функция `createAsyncThunk()` из модуля `@reduxjs/toolkit`:

`createAsyncThunk(<`*тип влияний*`>, <`*создающая функция*`>[, <`*настройки*`>=undefined])`

Тип влияний, обрабатываемых создаваемым переходником, указывается в виде строки.

Создающая функция должна принимать два параметра: значение параметра, переданного создателю влияний, который генерируется функцией `createAsyncThunk()` (о нем — позже), и простой объект с набором свойств, хранящих необходимые для

работы переходника данные (он по соглашению обычно имеет имя `thunkAPI` и будет подробно рассмотрен далее). *Функция* должна выполнять заданную асинхронную операцию, вырабатывающую требуемое значение (например, получающую данные от бэкенда).

Возвращать *создающая функция* должна промис, содержащий в качестве нагрузки значение, которое было вычислено в результате выполнения асинхронной операции, или само это значение. Более подробно возврат результата и обработка ошибок будут описаны далее.

Настройки указываются в очень специфических случаях и также рассмотрены далее.

Функция `createAsyncThunk()` возвращает сгенерированный создатель влияний с *типом*, указанным в первом параметре этой функции. Значение единственного параметра, передаваемого этому создателю влияний при вызове, и передается *создающей функции* в первом параметре.

Если создателю влияний требуется передавать несколько значений, их следует заключить в простой объект, который и передать в качестве параметра.

Вот пример создания переходника, помечающего дело с указанным идентификатором как выполненное и возвращающего идентификатор дела:

```
import { createAsyncThunk } from '@reduxjs/toolkit';
import { setDoneOnBackend } from './api.js';
import store from './store.js';
export const setDone = createAsyncThunk(
    'todos/setDone',
    async (key, thunkAPI) => {
        const user = store.getState().currentUser;
        await setDoneOnBackend(user.uid, key);
        return key;
    }
);
```

Теперь рассмотрим свойства объекта, передаваемого *создающей функции* вторым параметром:

- `dispatch` — ссылка на метод `dispatch()` хранилища Redux;
- `getState` — ссылка на метод `getState()` хранилища;
- `requestId` — уникальный идентификатор операции, запущенной переходником, в виде строки. Может применяться для предотвращения повторного запуска той же операции. Пример:

```
const someThunk = createAsyncThunk(
    'operations/runOperation',
    async (someData, { getState, requestId }) => {
        const state = getState();
        if (state.operations[requestId] !== 'running') {
            state.operations[requestId] = 'running';
```

```
            // Запускаем операцию
            state.operations[requestId] = 'finished';
        }
    }
);
```

- `signal` — сигнал-прерыватель, принадлежащий объекту встроенного класса `AbortController`, который генерируется автоматически при создании переходника. Обычно применяется для прерывания асинхронного запроса к бэкенду. Пример:

```
const getData = createAsyncThunk(
    'data/getData',
    async (stream, thunkAPI) => {
        return await fetch('/api/getdata', { signal: thunkAPI.signal });
    }
);
```

Сигнал-прерыватель также может использоваться для выяснения, была ли прервана извне операция, запущенная текущим переходником. Пример:

```
const readStream = createAsyncThunk(
    'streams/readStream',
    async (stream, { signal }) => {
        const reader = stream.getReader();
        let done = false, result = '';
        while (!done) {
            if (signal.aborted)
                throw new Error('Чтение прервано!');
            const read = await reader.read();
            result += read.value;
            done = read.done;
        }
        return result;
    }
);
```

- `rejectWithValue` — функция, используемая для оповещения о возникшей при выполнении переходника ошибке с выдачей указанного значения. Должна принимать единственный параметр — значение, передаваемое вызывающему коду. Пример:

```
export const setDone = createAsyncThunk(
    'todos/setDone',
    async (key, { rejectWithValue }) => {
        const user = store.getState().currentUser;
        try {
            const result = await setDoneOnBackend(user.uid, key);
            return key;
```

```
        } catch(err) {
            return rejectWithValue(err.code);
        }
    }
);
```

Настройки задаются в виде простого объекта со следующими свойствами:

- `condition` — функция, проверяющая, действительно ли необходимо запускать операцию, реализуемую переходником.

  Эта функция должна принимать два параметра: значение параметра, получаемого создателем влияния, который создается функцией `createAsyncThunk()`, и простой объект со свойством `getState`, хранящим ссылку на одноименный метод хранилища. Возвращать она должна `true`, если операцию требуется запустить, и `false` — чтобы отменить запуск операции.

  Пример:

```
const someThunk = createAsyncThunk(
    'operations/runOperation',
    async (someData, { getState }) => {
        const state = getState();
        state.operationStatus = 'running';
        // Запускаем операцию
    },
    {
        condition: (someData, { getState }) =>
            return getState().operationStatus !== 'running'
    }
);
```

  Если это свойство не указано, операция будет запускаться всегда;

- `dispatchConditionRejection` — принимается во внимание, если функция из свойства `condition` вернула `false`. Если этому свойству дано значение `false`, подавление запуска операции не будет обрабатываться никак. Если же свойству присвоено `true`, подавленная операция будет обработана таким же образом, как и завершившаяся неудачей.

15.2.5.2. Создание обработчиков операций

Переходники, создаваемые функцией `createAsyncThunk()`, лишь запускают выполнение асинхронных операций. Завершение этих операций, удачное или неудачное, обрабатывается особыми обработчиками, записываемыми в составе преобразователей.

Создатель влияний, возвращаемый функцией `createAsyncThunk()` и запускающий выполнение соответствующего переходника, содержит три свойства. Эти свойства хранят другие три создателя влияний, также сгенерированные автоматически:

- `pending` — автоматически вызывается сразу после запуска асинхронной операции. Создает влияние с типом формата:

  `<тип основного влияния>/pending`

  Тип основного влияния указывается в вызове функции `createAsyncThunk()`.

  Свойство `payload` созданного влияния хранит значение `undefined`;

- `fulfilled` — автоматически вызывается после успешного завершения операции и получения ее результата. Создает влияние с типом формата:

  `<тип основного влияния>/fulfilled`

  Свойство `payload` созданного влияния хранит результат операции;

- `rejected` — автоматически вызывается после неуспешного завершения асинхронной операции. Создает влияние с типом формата:

  `<тип основного влияния>/rejected`

  Объект созданного влияния, в зависимости от причины неуспешного выполнения операции, содержит следующие свойства:

  - возникновение ошибки — свойство `error` с объектом исключения, свойство `payload` — со значением `undefined`;
  - вызов функции из свойства `rejectWithValue` объекта `thunkAPI` (см. *разд. 15.2.5.1*) — свойство `payload` со значением, переданным упомянутой функции при вызове, и свойство `error` с простым объектом, содержащим свойство `message`, в котором хранится строка `'Rejected'`;

Например, при создании переходника, обрабатывающего влияние типа `todos/setDone`, будут созданы создатели влияния типов `todos/setDone/pending`, `todos/setDone/fulfilled` и `todos/setDone/rejected`.

Обработчики влияний этих трех типов добавляются в объект построителя в виде строгих условий — вызовами метода `addCase()` (подробности — в *разд. 15.2.2*).

Кроме того, в построитель можно добавить нестрогое условие, выполняющееся при завершении асинхронной операции — неважно, успешном или неуспешном. Такое условие добавляется вызовом метода `addMatcher()`. Функция проверки, задаваемая в первом параметре этого метода, хранится в свойстве `settled` создателя влияний.

Все объекты влияний дополнительно содержат свойство `meta`, в котором хранится простой объект с полезными сведениями о текущем влиянии. Этот объект поддерживает следующие свойства:

- `requestStatus` — обозначение статуса выполняемой операции в виде одной из строк: `'pending'` (выполняется), `'fulfilled'` (завершена успешно) или `'rejected'` (завершена неуспешно);
- `arg` — значение параметра, переданного создателю влияний при вызове;
- `requestId` — уникальный идентификатор операции, запущенной переходником, в виде строки.

Пример преобразователя:

```
import { setDone } from './api.js';

const reducer = createReducer(
    { status: '', todos: [], error: null, currentUser: null },
    builder => {
        .addCase(setDone.pending, state => {
            state.status = 'started';
        })
        .addCase(setDone.fulfilled, (state, { payload }) => {
            const deed = state.todos.find(
                current => current.key === payload
            );
            if (deed)
                deed.done = true;
            state.status = 'done';
            state.error = null;
        })
        .addCase(setDone.rejected, (state, action) => {
            state.status = 'error';
            state.error = action.payload;
        })
        .addMatcher(setDone.settled, (state, { meta }) => {
            console.log(`Операция завершена. Статус: ${meta.requestStatus}`);
        }
    }
});
```

Если эти влияния планируется обрабатывать в преобразователе среза, то их следует поместить в состав дополнительного преобразователя, задаваемого в свойстве `extraReducers` объекта с настройками, который передается функции `createSlice()` (см. *разд. 15.2.4*), например:

```
const todosReducer = createSlice({
    name: 'todos',
    initialState: { status: '', data: [], error: null },
    . . .
    extraReducers: builder => {
        builder
            .addCase(setDone.pending, (state, action) => { . . . })
            .addCase(setDone.fulfilled, (state, { payload }) => { . . . })
            .addCase(setDone.rejected, (state, action) => { . . . })
            .addMatcher(setDone.settled, (state, { meta }) => { . . . })
    }
});
```

15.2.5.3. Запуск переходников, получение результата и обработка ошибок

Запустить переходник, сформированный функцией `createAsyncThunk()`, и соответственно реализуемую им асинхронную операцию можно, применив сгенерированный при его формировании создатель влияний:

```
import { setDone } from './api.js';
. . .
const currentUser = store.getState().currentUser;
. . .
<button title="Пометить как выполненное"
        onClick={() => store.dispatch(setDone(item.key))}>
    &#9745;
</button>
```

Если запущенный переходник успешно выполнился, запустивший его создатель влияний самостоятельно отправит влияние с типом формата *<тип основного влияния>*`/fulfilled` и свойством `payload`, хранящим возвращенный переходником результат. Если же в коде переходника возникнет ошибка, будет отправлено влияние типа *<тип основного влияния>*`/rejected`, хранящее в свойстве `error` объект исключения. Об этом говорилось в *разд. 15.2.5.2*.

Обработать оба упомянутых влияния с получением результата или исправлением возникшей ошибки можно в обработчике, являющемся частью написанного ранее преобразователя (см. тот же *разд. 15.2.5.2*).

Однако получить результат выполнения переходника или обработать возникшую в нем ошибку можно и непосредственно в месте его запуска.

Метод `dispatch()` хранилища при вызове переходника возвращает промис, содержащий одно из упомянутых ранее влияний. Вот пример извлечения результата успешно выполненной операции и обработки ошибки при ее неуспешном выполнении:

```
const result = await store.dispatch(setDone(itemKey));
if (result.payload) {
    // Операция выполнилась успешно
    // Получаем результат, возвращенный переходником
    const payload = result.payload;
    // Что-либо делаем с ним
} else {
    // Операция выполнилась неуспешно
    // Получаем объект исключения
    const err = result.error;
    // Что-либо делаем с возникшим исключением
}
```

Промис, возвращенный методом `dispatch()`, поддерживает дополнительный метод `unwrap()`. Этот метод, в зависимости от результата выполнения операции:

- успех — извлекает и возвращает значение свойства `payload` влияния;
- неуспех — извлекает из свойства `error` влияния исключение и возбуждает его.

Пример:

```
try {
    const result = await store.dispatch(setDone(itemKey)).unwrap();
    // Операция увенчалась успехом
} catch(err) {
    // Операция не увенчалась успехом
}
```

15.2.5.4. Прерывание выполнения операций

Прервать выполнение асинхронной операции можно, вызвав у промиса, возвращенного методом `dispatch()` хранилища, метод `abort()`.

В результате будет автоматически отправлено влияние с типом *<тип основного влияния>*`/rejected` и исключением типа `AbortError` в свойстве `error`.

Вот пример запуска загрузки данных с бэкенда при монтировании компонента и прерывания ее при размонтировании:

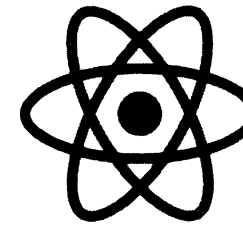
```
export default function TodoList() {
    . . .
    useEffect(() => {
        const promise = store.dispatch(getTodos());
        return () => {promise.abort()};
    }, []);
    . . .
}
```

15.2.6. Создание мемоизированных селекторов

При установке библиотеки Redux Toolkit также устанавливается библиотека Reselect (см. *разд. 14.1.10.3*). Функция `createSelector()`, объявленная в этой библиотеке, реэкспортируется модулем `@reduxjs/toolkit`. Так что при необходимости создать мемоизированный селектор можно импортировать функцию именно из этого модуля. Пример:

```
import { createSelector } from '@reduxjs/toolkit';
const selectItem = createSelector( . . . );
```

ГЛАВА 16



Библиотека Formik: создание веб-форм

Код, создающий веб-формы в React-компонентах, зачастую получается очень большим и сложным, а писать его весьма трудоемко (особенно если требуется реализовать валидацию введенных данных).

Дополнительная библиотека Formik существенно упрощает разработку веб-форм. Установка этой библиотеки выполняется командой:

```
npm install formik
```

> ***НА ЗАМЕТКУ***
>
> Полное описание библиотеки Formik приведено по интернет-адресу: **https://formik.org/**.

Все сущности — компоненты и хуки, — входящие в состав этой библиотеки, объявлены в модуле `formik`.

16.1. Быстрое создание веб-форм

Для быстрого создания веб-форм библиотека Formik предоставляет набор удобных в использовании компонентов.

16.1.1. Компонент *Formik*: построитель веб-форм

Компонент `Formik` реализует функциональность *построителя веб-форм* — сущности, которая обеспечивает работу веб-формы: занесение введенных значений во внутреннее хранилище, их валидацию, обработку нажатий кнопок отправки данных и сброса.

Компонент `Formik` принимает следующие основные пропы:

- `initialValues` — начальные значения, которые должны быть занесены в элементы управления сразу после вывода веб-формы.

  Значением пропа должен быть простой объект со свойствами, имена которых должны совпадать с наименованиями элементов управления (наименования за-

даются атрибутом тега `name`), а значения, собственно, и будут занесены в элементы управления.

> **Внимание!**
>
> Следует задать начальные значения для *всех* элементов управления, присутствующих в веб-форме. Наличие элемента управления без начального значения вызовет критическую ошибку.

- `enableReinitialize` — если `true`, веб-форма будет сбрасываться при изменении изначальных значений, указанных в пропе `initialValues`, если `false` — не будет (по умолчанию: `false`);
- `initialStatus` — изначальный статус. *Статусом веб-формы* в терминологии Formik называются произвольные данных, сохраняемые непосредственно в веб-форме. Статус указывается в виде простого объекта с произвольным набором свойств, хранящих произвольные значения. По умолчанию: `undefined`.

Компонент `Formik` должен содержать единственный потомок — веб-форму, которую он обслуживает. Веб-форму можно указать:

- непосредственно в теге компонента `Formik`:

```
import { Formik, Form } from 'formik';
. . .
<Formik . . . >
    <Form>
        // Код, создающий элементы управления
    </Form>
</Formik>
```

- в пропе `component` — в виде ссылки на компонент, реализующий саму веб-форму:

```
function MyForm(props) {
    return (
        <Form>
            // Код, создающий элементы управления
        </Form>
    );
}
. . .
<Formik . . . component={MyForm} />
```

Более подробно об этом способе указания веб-формы рассказано в *разд. 16.2*.

> **Внимание!**
>
> Если веб-форма указана и в теге компонента `Formik`, и в пропе `component`, предпочтение отдается веб-форме из пропа `component`.

> **За кулисами**
>
> Построитель веб-форм создает собственный контекст, в который записывает объект, предназначенный для хранения введенных значений и методов, предназначенных использоваться в качестве обработчиков события `submit` веб-формы, событий `change` и

blur элементов управления (*объект хранения*). Далее построитель привязывает эти обработчики к событиям веб-формы и элементов управления и выводит в элементах управления значения, содержащиеся в свойствах объекта, тем самым реализуя обработку этих элементов в стиле React (см. *разд. 8.1*).

16.1.2. Компонент *Form*: веб-форма

Сама веб-форма, размещаемая внутри построителя веб-форм, реализуется компонентом Form. Никаких специфических пропов он не поддерживает.

В компонент Form помещаются компоненты, создающие отдельные элементы управления, равно как и HTML-теги, которые формируют всевозможные вспомогательные элементы: надписи, блоки разметки и др. Вот пример кода, создающего веб-форму для ввода нового дела:

```
import { Formik, Form, Field } from 'formik';
. . .
<Formik initialValues={{ title: '', desc: '' }}>
    <Form>
        <label>Заголовок</label>
        <Field name="title" />
        <label>Примечание</label>
        <Field as="textarea" name="desc" />
        <input type="submit" value="Создать дело" />
    </Form>
</Formik>
```

16.1.3. Компонент *Field*: элемент управления

Компонент Field создает отдельный элемент управления указанного типа: поле ввода, флажок, переключатель, область редактирования, список, регулятор и пр. С помощью этого компонента могут создаваться элементы управления, реализуемые как средствами HTML, так и другими React-компонентами.

Компонент Field поддерживает следующие специфические пропы:

- name — наименование элемента управления HTML, создаваемого компонентом, и одновременно имя свойства в объекте хранения для записи значения, занесенного в этот элемент;
- as — разновидность создаваемого элемента управления в виде:
  - имени HTML-тега, создающего элемент управления, в виде строки.

    Пример создания области редактирования:

    ```
    <Field as="textarea" name="desc" />
    <Field as="select" name="kind" />
    ```

  - ссылки на React-компонент, реализующий элемент управления:

    ```
    <Field as={SuperTextArea} name="desc" />
    ```

  Значение по умолчанию: "input";

- `type` — принимается во внимание, если в качестве разновидности создаваемого элемента управления указано `"input"`, и задает тип элемента:

```
<Field type="password" name="userPassword" />
```

  Значение по умолчанию: `"text"`;

- `innerRef` — используется для создания функционального рефа, ссылающегося на созданный компонентом элемент управления HTML:

```
let titleRef = useRef(undefined);
. . .
<Field name="title" innerRef={el => {titleRef.current = el}} . . . />
. . .
// Устанавливаем фокус ввода на поле для занесения заголовка дела
titleRef.current.focus();
```

- любые другие пропы — будут переданы элементу управления, создаваемому компонентом:

```
<Field name="title" className="input" required />
```

В теге `<Field>` можно указать содержимое, необходимое для создания элемента управления — например, набор пунктов списка.

Пример использования этого компонента приведен в *разд. 16.1.2*.

16.1.3.1. Вывод флажков, переключателей, списков и полей выбора файлов

- Флажок:
  - если проп `value` не указан — в соответствующее свойство объекта хранения заносится состояние флажка в виде логической величины `true` (флажок установлен) или `false` (сброшен):

```
<Formik initialState={{ loveReact: true }}>
    <Form>
        <Field type="checkbox" name="loveReact" /> Я люблю React
    </Form>
</Formik>
```

  - если проп `value` указан — в свойство объекта хранения будет занесен массив с единственным элементом — значением пропа `value`, если флажок установлен, и пустой массив — если флажок сброшен:

```
<Formik initialState={{ loveReact: ['yes'] }}>
    <Form>
        <Field type="checkbox" name="loveReact" value="yes" />
        Я люблю React
    </Form>
</Formik>
```

Эта особенность Formik позволяет создавать наборы флажков, состояния которых хранятся в одном массиве в виде отдельных элементов. Для этого

достаточно у всех флажков набора указать одинаковое наименование (в пропе name) и разные значения (в пропе value). Пример:

```
<Formik initialState={{ love: ['React', 'Django'] }}>
    <Form>
        Ваши любимые веб-фреймворки:<br />
        <Field type="checkbox" name="love" value="React" />
        React<br />
        <Field type="checkbox" name="love" value="Angular" />
        Angular<br />
        <Field type="checkbox" name="love" value="Laravel" />
        Laravel<br />
        <Field type="checkbox" name="love" value="Django" />
        Django
    </Form>
</Formik>
```

- набор переключателей — в свойство объекта хранения будет занесено значение установленного переключателя (задается в пропе value). При этом у всех переключателей, входящих в набор, должно быть задано одинаковое наименование (в пропе name). Пример:

```
<Formik initialState={{ prefer: 'React' }}>
    <Form>
        Какой веб-фреймворк вы предпочтете?<br>
        <Field type="radio" name="prefer" value="React" />   React<br />
        <Field type="radio" name="prefer" value="Angular" /> Angular<br />
        <Field type="radio" name="prefer" value="Vue" />     Vue
    </Form>
</Formik>
```

Если требуется, чтобы изначально ни один переключатель набора не был установлен, в соответствующее свойство объекта хранения следует занести пустую строку:

```
<Formik initialState={{ prefer: '' }}>
    . . .
</Formik>
```

- список:
  - позволяющий выбрать лишь один пункт — в свойство объекта хранения будет занесено значение выбранного пункта (задается в пропе value):

```
<Formik initialState={{ love: 'React' }}>
    <Form>
        Ваш любимый веб-фреймворк:<br />
        <Field as="select" name="love">
            <option value="React">React</option>
            <option value="Angular">Angular</option>
            <option value="Vue">Vue</option>
        </Field>
```

```
    </Form>
</Formik>
```

Если требуется, чтобы изначально ни один пункт в списке не был выбран, в соответствующее свойство объекта хранения следует занести пустую строку;

- позволяющий одновременно выбрать несколько пунктов — в свойство объекта хранения будет занесен массив со значениями всех выбранных пунктов:

```
<Formik initialState={{ prefer: ['React', 'Django'] }}>
    <Form>
        Какой веб-фреймворк вы предпочтете?<br />
        <Field as="select" name="prefer" size="4" multiple>
            <option value="React">React</option>
            <option value="Angular">Angular</option>
            <option value="Laravel">Laravel</option>
            <option value="Django">Django</option>
        </Field>
    </Form>
</Formik>
```

Если требуется, чтобы изначально ни один пункт в списке не был выбран, в соответствующее свойство объекта хранения следует занести пустой массив;

- поле выбора файла — в свойство объекта хранения будет занесено имя выбранного файла (или имя последнего из выбранных файлов, если поле предоставляет множественный выбор) с фиктивным путем.

  К сожалению, компонент `Field` не позволяет создать полноценное поле выбора файла, предоставляющее для извлечения выбранный файл (или коллекцию выбранных файлов). Реализовать такое поле можно, лишь написав его самостоятельно (как это сделать, будет рассказано в конце главы).

16.1.4. Обработка занесенных данных

Обработка данных, занесенных в веб-форму, практически всегда выполняется, когда веб-форма производит отправку данных. Этот процесс запускается нажатием кнопки отправки данных, присутствующей в веб-форме.

Компонент `Formik` поддерживает проп `onSubmit`, в котором указывается функция, вызываемая, когда веб-форма производит отправку данных. В теле этой функции и будет производиться обработка введенных данных.

Функция, обрабатывающая данные, должна принимать два параметра:

- ссылку на объект хранения (традиционно именуется `values`).

  Этот объект содержит набор свойств, имена которых совпадают с наименованиями элементов управления, находящихся в веб-форме. Свойства хранят значения, занесенные в соответствующие элементы управления;

- объект, содержащий все методы, описанные в *разд. 16.2*.

Результата эта функция возвращать не должна.

Сразу после запуска отправки данных веб-форма помечается как производящая отправку. Благодаря этому можно без особого труда реализовать предотвращение повторной отправки данных (как это сделать, рассказано позже).

После обработки данных, если не выполняется навигация на какой-либо другой экран, следует явно пометить веб-форму как завершившую отправку данных. Для этого надо у объекта, полученного вторым параметром, вызвать метод `setSubmitting()`, передав ему параметром значение `false`.

Если после обработки данных производится навигация на другой экран, помечать веб-форму как завершившую отправку данных необязательно.

Вот пример обработки данных, занесенных в веб-форму создания нового дела:

```
export default function TodoAdd(props) {
    . . .
    const handleFormSubmit = (values, formAPI) => {
        const date = new Date();
        const newDeed = {
            ...values,
            done: false,
            createdAt: date.toLocaleString()
        };
        props.add(newDeed);
        formAPI.setSubmitting(false);
    };

    return (
        <Formik . . . onSubmit={handleFormSubmit}>
            . . .
        </Formik>
    );
}
```

16.1.5. Валидация данных

Библиотека Formik значительно упрощает как саму валидацию данных, так и вывод сообщений об ошибках ввода.

Компонент-построитель веб-форм `Formik` поддерживает проп `validate`. Ему присваивается функция, которая и выполнит валидацию данных. Она должна принимать единственный параметр — ссылку на объект хранения. Возвращать она должна простой объект, содержащий свойства, имена которых совпадают с наименованиями элементов управления, а значениями являются сообщения об ошибках ввода в эти элементы. Если в какой-либо элемент управления занесено корректное значение, соответствующему свойству нужно дать значение `undefined` или вообще не указывать в объекте это свойство.

Функция, валидирующая данные, также может вернуть промис, содержащий аналогичный объект с сообщениями об ошибках в качестве нагрузки.

Вот пример кода, валидирующего сведения о создаваемом деле:

```
export default function TodoAdd(props) {
    . . .
    const validate = values => {
        const errors = {};
        if (!values.title)
            errors.title = 'Укажите заголовок дела';
        else if (values.title.length > 100)
            errors.title= 'Заголовок дела должен быть не длиннее 100 символов';
        return errors;
    };

    return (
        <Formik . . . validate={validate}>
            . . .
        </Formik>
    );
}
```

Для вывода сообщения об ошибке ввода в элемент управления служит компонент `ErrorMessage`. Он поддерживает следующие пропы:

- `name` — наименование элемента управления;
- `component` — тип элемента, в котором должно быть заключено сообщение, в виде:
  - имени HTML-тега, заданного в виде строки;

    ```
    import { ErrorMessage } from 'formik';
    . . .
    <label>Заголовок</label>
    <Field name="title" />
    <ErrorMessage name="title" component="div" />
    ```

  - ссылки на React-компонент.

  Если проп не указан, сообщение будет выведено без заключения в какой бы то ни было тег или компонент;
- `render` — позволяет вывести сообщение об ошибке произвольным образом.

  Значением пропа должна быть функция, принимающая единственным параметром сообщение об ошибке и возвращающая фрагмент содержимого, который будет помещен на страницу. Пример вывода сообщения с оформлением в стиле CSS-фреймворка Bulma:

  ```
  <ErrorMessage name="title"
        render={msg => <p className="help is-danger">{msg}</p>} />
  ```

  Следует указывать либо проп `component`, либо проп `render`, но не оба сразу.

Компонент `ErrorMessage` выводит сообщение об ошибке только в том случае, если соответствующий элемент управления является посещенным. *Посещенным* счита-

ется тот элемент управления, на который посетитель поместил фокус ввода и либо ввел что-либо в этот элемент, либо пропустил его и последовал дальше. У непосещенных элементов управления сообщения об ошибках не выводятся, т. к. это может обескуражить посетителя.

Наиболее часто валидация данных реализуется на уровне построителя веб-форм, как было показано ранее. Однако есть возможность реализовать валидацию на уровне отдельного элемента управления. Это может пригодиться в случаях, если требуется валидировать значение, заносимое лишь в один элемент управления.

Функция, валидирующая значение, указывается в пропе `validate` компонента `Field`. Она должна принимать единственным параметром значение из элемента управления и возвращать:

- сообщение об ошибке ввода в виде строки — если при вводе была допущена ошибка.

  Также можно вернуть промис, содержащий сообщение об ошибке в качестве нагрузки;
- значение `undefined` — если занесено корректное значение.

  Вместо этого можно вообще не возвращать никакого результата.

Пример:

```
export default function TodoAdd(props) {
    . . .
    const validateTitle => title => {
        if (!title)
            return 'Укажите заголовок дела';
        else if (title.length > 100)
            return 'Заголовок дела должен быть не длиннее 100 символов';
    };
    return (
        <Formik . . . >
            <Form>
                <label>Заголовок</label>
                <Field name="title" validate={validateTitle} />
                <ErrorMessage name="title" component="div" />
                . . .
            </Form>
        </Formik>
    );
}
```

Компонент `Formik` поддерживает ряд пропов, управляющих процессом валидации:

- `validateOnChange` — если `true`, валидация будет выполняться при возникновении в элементе управления события `change`, если `false` — не будет выполняться (по умолчанию: `true`);

- `validateOnBlur` — если `true`, валидация будет выполняться при возникновении в элементе управления события `blur`, если `false` — не будет выполняться (по умолчанию: `true`);
- `validateOnMount` — если `true`, валидация будет выполняться при монтировании компонента и после изменения начальных значений, заданных в пропе `initialValues`, если `false` — не будет выполняться (по умолчанию: `false`).

И еще два пропа, которые могут пригодиться в специфических случаях:

- `initialErrors` — простой объект с изначальными сообщениями об ошибках ввода, которые будут выведены сразу после монтирования компонента. Должен иметь ту же структуру, что и объект, возвращаемый функцией валидации из пропа `validate`.

  Если проп не указан или в качестве его значения задан пустой объект, никаких сообщений об ошибках ввода изначально выведено не будет;
- `initialTouched` — сведения о том, какие элементы управления должны быть изначально помечены как посещенные. Значением должен быть простой объект со свойствами, имена которых совпадают с наименованиями элементов управления и которые хранят значения `true` (элемент управления должен быть помечен как посещенный) или `false` (как непосещенный).

  Если проп не указан или в качестве его значения задан пустой объект, все элементы управления будут помечены как непосещенные.

Эти пропы могут использоваться, если в веб-форме изначально выводятся заведомо некорректные данные, предназначенные для исправления. Пример:

```
<Formik initialValues={{ title: '', desc: 'И закончить приложение поскорее' }}
        initialErrors={{ title: 'Укажите заголовок дела' }}
        initialTouched={{ title: true }}
        . . .
>
    . . .
</Formik>
```

16.1.6. Ввод данных в массивы и объекты

Иногда элементы управления используются для ввода значений элементов какого-либо массива фиксированного размера или свойств какого-либо объекта фиксированной структуры. Formik может записывать вводимые значения непосредственно в массив или объект.

- Для занесения значения в элемент массива, имеющий указанный *индекс*, — следует указать в пропе `name` элемента управления конструкцию формата:

  <наименование элемента управления> [*<индекс элемента массива>*]

  или:

  <наименование элемента управления>.*<индекс элемента массива>*

- для занесения значения в свойство объекта, имеющее заданное *имя*, — следует указать в пропе `name` конструкцию формата:

  `<наименование элемента управления>.<имя свойства объекта>`

Пример:

```
<Formik initialValues={{
    name: { name1: '', name2: '' },
    phones: ['', '', '']
}} . . . >
    <Form>
        <label>Имя:</label>              <Field name="name.name1" />
        <label>Фамилия:</label>          <Field name="name.name2" />
        <label>Телефон домашний:</label> <Field name="phones[0]" />
        <label>Телефон рабочий:</label>  <Field name="phones[1]" />
        <label>Телефон сотовый:</label>  <Field name="phones.2" />
    </Form>
</Formik>
```

16.2. Создание более сложных веб-форм

Описанные в *разд. 16.1* программные инструменты Formik позволяют быстро создать типовую веб-форму, пригодную для большинства случаев. Однако если необходимо разработать более сложную веб-форму с дополнительной функциональностью (например, выводящую дополнительное поле ввода при установке флажка или каким-либо образом помечающую поле ввода в зависимости от занесенного в него значения), понадобятся более развитые и низкоуровневые инструменты.

В качестве содержимого тега `<Formik>` можно указать функцию, которая будет возвращать требуемую веб-форму, описанную языком JSX, и должна принимать в качестве единственного параметра объект построителя веб-форм. Пример:

```
<Formik . . . >
    {formik => (
        <Form>
            . . .
        </Form>
    )}
</Formik>
```

Этот объект предоставляет набор свойств, методов и обработчиков событий, с помощью которых реализуется функциональность создаваемой веб-формы.

Построитель веб-форм поддерживает следующие:

- свойства, доступные только для чтения:
  - `values` — простой объект со значениями, занесенными в элементы управления;

- `errors` — простой объект с сообщениями об ошибках, возвращенный функцией из пропа `validate` (*см. разд. 16.1.5*);
- `touched` — сведения о том, какой элемент управления является посещенным, а какой — непосещенным. Представлены в виде простого объекта со свойствами, имена которых совпадают с наименованиями элементов управления, и которые хранят значения `true` (элемент является посещенным) или `false` (является непосещенным);
- `status` — статус веб-формы;
- `isSubmitting` — `true`, если в текущий момент выполняется процесс отправки данных, `false` — в противном случае;
- `isValidating` — `true`, если в текущий момент выполняется валидация введенных данных, `false` — в противном случае;
- `isValid` — `true`, если все занесенные в веб-форму значения корректны, `false` — в противном случае;
- `dirty` — `true`, если хотя бы одно из введенных значений не совпадает с изначально заданным в пропе `initialValues` (т. е. посетитель ввел в веб-форму что-либо), `false` — если все значения совпадают с изначальными;
- `submitCount` — количество попыток отправить введенные в веб-форму данные. Увеличивается при каждой попытке отправки данных, уменьшается при каждом сбросе веб-формы;

♦ методы:

- `setFieldValue()` — заносит указанное *значение* в свойство объекта хранения, соответствующее элементу управления с заданным *наименованием*:

```
setFieldValue(<наименование>, <значение>[,
                              <выполнить валидацию?>=true])
```

Если параметру *выполнить валидацию* дать значение `true`, новое *значение* после занесения будет подвергнуто валидации, если дать `false` — не будет.

Если параметру *выполнить валидацию* дано значение `true`, метод возвращает промис. Если заносимое *значение* корректно, промис в качестве нагрузки содержит значение `undefined`, в противном случае — сообщение об ошибке ввода.

Если параметру *выполнить валидацию* дано значение `false`, метод не возвращает результата;

- `setValues()` — заносит новые *значения* сразу в несколько свойств объекта хранения:

```
setValues(<значения>[, <выполнить валидацию?>=true])
```

Значения указываются в виде простого объекта, имена свойств которого совпадают с наименованиями элементов управления, а сами свойства хранят новые значения, подлежащие занесению в объект хранения.

Если параметру *выполнить валидацию* дать значение `true`, заносимые *значения* будут подвергнуты валидации, если дать значение `false` — не будут.

Если параметру *выполнить валидацию* дано значение `true`, метод возвращает промис. Если все заносимые *значения* корректны, промис содержит значение `undefined`, в противном случае — простой объект с сообщениями об ошибках ввода.

Если параметру *выполнить валидацию* дано значение `false`, метод не возвращает результата.

Эти методы могут понадобиться при реализации собственной функциональности занесения значений в объект хранения взамен обработчика события `change`, предоставляемого построителем веб-форм (обработчик описан далее);

- `validateForm()` — выполняет валидацию сразу нескольких значений. Поддерживает два параметра вызова:

```
validateForm()
validateForm([<значения>])
```

Первый формат выполняет валидацию данных, занесенных в элементы управления веб-формы.

Второй формат валидирует заданные *значения*. Они должны быть указаны в виде простого объекта со свойствами, имена которых совпадают с наименованиями элементов управления и которые хранят значения, подлежащие валидации.

Метод возвращает промис, подтверждаемый по окончании валидации и хранящий простой объект с сообщениями об ошибках ввода;

- `validateField(<наименование>)` — валидирует значение из элемента управления с заданным *наименованием*.

Эти методы пригодятся для проведения принудительной валидации веб-формы;

- `setFieldError()` — задает *сообщение об ошибке ввода* у элемента управления с указанным *наименованием*:

```
setFieldError(<наименование>, <сообщение об ошибке ввода>)
```

И *наименование*, и *сообщение* задаются в виде строк;

- `setErrors(<сообщения об ошибках ввода>)` — задает *сообщения об ошибках ввода* сразу у нескольких элементов управления. *Сообщения* указываются в виде простого объекта, аналогичного возвращаемому функцией валидации, которая задается в пропе `validate` (см. *разд. 16.1.5*).

Эти методы могут пригодиться при реализации какой-либо специфической валидации введенных значений;

- `setFieldTouched()` — помечает элемент управления с заданным *наименованием* как посещенный или непосещенный:

```
setFieldTouched(<наименование>[, <посещенный?>=true[,
                                <выполнить валидацию?>=true]])
```

Если параметр *посещенный* имеет значение `true`, элемент управления будет помечен как посещенный, если `false` — как непосещенный.

Назначение параметра *выполнить валидацию* и возвращаемый результат те же, что и у метода `setFieldValue()`;

- `setTouched()` — помечает сразу несколько элементов управления как посещенные или непосещенные:

```
setTouched(<состояние элементов управления>[,
           <выполнить валидацию?>=true])
```

Состояние должно представлять собой простой объект, имена свойств которого совпадают с наименованиями элементов управления, а сами свойства хранят значения `true` (элемент управления требуется сделать посещенным) или `false` (непосещенным).

Назначение параметра *выполнить валидацию* и возвращаемый результат те же, что и у метода `setValues()`.

Эти методы могут понадобиться при реализации собственных обработчиков события `blur` взамен предоставляемого построителем веб-форм (который описан далее);

- `setStatus(<статус>)` — задает у веб-формы указанный *статус*;
- `submitForm()` — запускает процесс отправки данных. Возвращает промис, хранящий значение `undefined`, подтверждаемый при успешной отправке и отклоняемый, если веб-форма не прошла валидацию.

  Этот метод пригодится, если отправка данных запускается не специализированной кнопкой, а иным образом (например, сразу после выбора пункта в списке);

- `resetForm()` — выполняет сброс веб-формы;
- `setSubmitting()` — переводит веб-форму в состояние отправки данных или обычное состояние:

```
setSubmitting(<состояние отправки данных?>)
```

Значение `true`, переданное методу, переводит веб-форму в состояние отправки данных, значение `false` — в обычное состояние.

Метод может пригодиться при реализации своей собственной функциональности отправки данных вместо предлагаемой построителем веб-форм;

- обработчики событий:
  - `handleChange()` — обработчик события `change` элементов управления. Копирует значения, занесенные в элементы управления, в свойства объекта хранения;

- `handleBlur()` — обработчик события `blur` элементов управления. Благодаря ему Formik может отследить, какой элемент является посещенным;
- `handleSubmit()` — обработчик события `submit` веб-формы. Запускает процесс валидации и последующей отправки данных;
- `handleReset()` — обработчик события `reset` веб-формы. Заполняет элементы управления изначальными значениями.

При использовании описанных здесь программных инструментов элементы управления создаются с применением обычных HTML-тегов: `<input>`, `<textarea>`, `<select>` и др. У всех этих тегов необходимо указать наименование, записав его в атрибуте `name`.

Пример сложной веб-формы:

```
<Formik initialValues={{ title: '', desc: '' }} onSubmit={handleFormSubmit}>
    {formik => (
        <form onSubmit={formik.handleSubmit} onReset={formik.handleReset}>
            <label>Заголовок</label>
            <input name="title" value={formik.values.title}
                   className={formik.touched.title ? 'touched-input' : ''}
                   onChange={formik.handleChange}
                   onBlur={formik.handleBlur} />
            {formik.touched.title && formik.errors.title ? (
                <p>{formik.errors.title}</p>
            ) : null}
            <label>Примечание</label>
            <textarea name="desc" value={formik.values.desc}
                      className={formik.touched.desc ? 'touched-textarea' : ''}
                      onChange={formik.handleChange}
                      onBlur={formik.handleBlur} />
            {formik.touched.desc && formik.errors.desc ? (
                <p>{formik.errors.desc}</p>
            ) : null}
            <input type="submit" value="Создать дело" />
        </form>
    )}
</Formik>
```

Пример использования метода `resetForm()` для выполнения сброса веб-формы по нажатию обычной кнопки:

```
{formik => (
    <form onSubmit={formik.handleSubmit}>
        . . .
        <input type="button" value="Сброс" onClick={formik.resetForm} />
    </form>
)}
```

Писать код, выводящий в элементе управления значение из свойства объекта хранения и привязывающий к элементу обработчики событий `change` и `blur`, достаточ-

но трудоемко. Его можно заменить вызовом метода `getFieldProps(<наименование>)`, которому передается *наименование* элемента управления. Метод возвращает простой объект со всем необходимыми пропами, которые нужно лишь передать элементу управления. Пример:

```
<textarea name="desc" { ...formik.getFieldProps('desc') } />
```

Наконец, можно создать компонент, реализующий саму веб-форму, и указать его в пропе `component` построителя веб-форм. Указанному компоненту будут переданы в качестве пропов все свойства и методы объекта построителя веб-форм: `values`, `errors`, `handleSubmit()`, `handleReset()` и др. Пример:

```
<Formik component={MyForm} . . . />
. . .
function MyForm(props) {
    return (
        <form onSubmit={props.handleSubmit} onReset={props.handleReset}>
            <label>Заголовок</label>
            <input name="title" value={props.values.title}
                   onChange={props.handleChange}
                   onBlur={props.handleBlur} />
            {props.touched.title && props.errors.title ? (
                <p>{props.errors.title}</p>
            ) : null}
            . . .
        </form>
    );
}
```

16.3. Создание своих элементов управления

К сожалению, библиотека Formik некорректно обрабатывает поля для выбора файлов — вместо коллекции выбранных файлов такое поле помещает в объект хранения имя выбранного файла, совершенно бесполезное.

Однако Formik позволяет создавать на основе компонента `Field` произвольные элементы управления с произвольными интерфейсом и логикой.

В тег `<Field>` помещается функция, которая должна возвращать в качестве результата содержимое создаваемого элемента управления, написанное на JSX. Эта функция также должна принимать единственный параметр — простой объект со следующими свойствами:

- `field` — простой объект, содержащий свойства:
  - `name` — наименование элемента управления, заданное одноименным пропом в самом компоненте;
  - `value` — текущее значение элемента управления, содержащееся в объекте хранения;

- onChange — обработчик события change;
- onBlur — обработчик события blur;

♦ meta — простой объект со свойствами:

- error — сообщение об ошибке ввода в текущий элемент управления или undefined, если введено корректное значение;
- touched — true, если текущий элемент управления является посещенным, и false — если является непосещенным;
- initialValue — изначальное значение элемента управления;
- initialError — изначальное сообщение об ошибке или undefined, если оно не было задано;
- initialTouched — true, если элемент управления изначально был помечен как посещенный, и false — если он изначально непосещенный;

♦ form — простой объект, который содержит свойства и методы, поддерживаемые построителем веб-форм и описанные в *разд. 16.2*.

Вот пример создания корректно работающего поля для выбора файлов:

```
<Field name="image">
    {({ field, form, meta }) => (
        <input type="file" name={field.name} accept="image/*"
               onChange={() => {
                   const cFiles = evt.target.files;
                   if (cFiles.length > 0) {
                       const fileReader = new FileReader();
                       fileReader.onload = () => {
                           form.setFieldValue(field.name, fileReader.result);
                       };
                       fileReader.readAsDataURL(cFiles[0]);
                   } else
                       form.setFieldValue(field.name, '');
               }}
               onBlur={() => {form.setFieldTouched(field.name)}}
        />
    )}
</Field>
```

16.4. Сброс веб-форм

При выполнении сброса веб-формы нажатием кнопки сброса в элементы управления будут подставлены значения, указанные в пропе initialValues построителя веб-форм (компонента Formik). Это поведение библиотеки по умолчанию, и оно приемлемо в большинстве случаев.

Однако есть возможность при сбросе веб-формы указать другие значения, заносимые в элементы управления, а также сообщения об ошибках ввода, статус и другие

данные. Для этого следует в пропе `onReset` построителя веб-форм задать функцию, которая будет вызываться при сбросе и выполнять все необходимые действия. Функция должна принимать те же параметры, что и указываемая в пропе `onSubmit` (см. *разд. 16.1.4*), и не должна возвращать результата.

Вот пример сброса с подстановкой в поле ввода заголовка строки формата «дело *<порядковый номер>*»:

```
export default function TodoAdd() {
    let cnt = 0;
    . . .
    return (
        <Formik . . . onReset={values => {values.title = `Дело ${++cnt}`}}>
            . . .
        </Formik>
    );
}
```

Метод `resetForm()`, описанный в *разд. 16.2*, поддерживает альтернативный формат вызова:

```
resetForm(<состояние веб-формы после сброса>)
```

Состояние указывается в виде простого объекта со следующими свойствами:

- `values` — значения для занесения в элементы управления. Указываются в том же формате, что и в вызове метода `setValues()` (см. *разд. 16.2*);
- `errors` — сообщения об ошибках ввода. Указываются в том же формате, что и в вызове метода `setErrors()`;
- `touched` — признаки, считать ли тот или иной элемент управления посещенным. Указываются в том же формате, что и в вызове метода `setTouched()`;
- `isSubmitting` — `true`, чтобы перевести веб-форму в состояние отправки данных, и `false`, чтобы перевести ее в обычное состояние;
- `isValidating` — `true`, чтобы перевести веб-форму в состояние валидации, и `false`, чтобы перевести ее в обычное состояние;
- `status` — новый статус веб-формы;
- `submitCount` — новое количество попыток выполнить отправку данных.

Все эти свойства являются необязательными к указанию.

Пример сброса веб-формы с простым занесением в нее новых данных:

```
formik.resetForm({
    values: {
        title: 'Продолжить работу над веб-приложением',
        desc: 'Пока оно не будет завершено'
    }
});
```

Пример сброса с отображением сообщений об ошибках ввода:

```
formik.resetForm({
    values: {
        title: '',
        desc: 'И закончить приложение поскорее'
    },
    errors: {
        title: 'Введите заголовок дела'
    },
    touched: {
        title: true
    },
    submitCount: 1
});
```

16.5. Массивы элементов управления

В *разд. 16.1.6* описывалось использование компонентов Formik для занесения значений в элементы массива фиксированного размера. Однако чаще посетители посредством веб-форм заполняют массивы неограниченного размера, добавляя в них новые элементы и удаляя ненужные.

Для этих случаев Formik предоставляет *массивы элементов управления*. Такой массив содержит набор элементов управления, каждый из которых выводит значение одного из элементов массива данных, а также средства для добавления и удаления элементов.

Массив элементов управления представляется компонентом `FieldArray`. Он поддерживает следующие пропы:

- `name` — наименование массива элементов управления и одновременно имя свойства объекта хранения, в котором будет храниться массив с данными;
- `validateOnChange` — если `true`, валидация будет выполняться после любого изменения, сделанного в массиве данных (включая добавление и удаление элементов), если `false` — не будет выполняться (по умолчанию: `true`).

В теге `<FieldArray>` указывается функция, которая должна возвращать массив элементов управления, предназначенный для работы с массивом данных. В качестве единственного параметра эта функция должна принимать простой объект, содержащий методы, посредством которых можно манипулировать массивом данных. Вот эти методы:

- `push(<`*элемент*`>)` — добавляет в конец массива данных указанный новый *элемент*;
- `unshift(<`*элемент*`>)` — добавляет в начало массива данных указанный новый *элемент* и возвращает новый размер массива;
- `insert(<`*индекс*`>, <`*элемент*`>)` — вставляет в массив данных новый *элемент*, размещая его по указанному *индексу*;

- `replace(<индекс>, <элемент>)` — заменяет элемент массива данных, находящийся по указанному *индексу*, заданным *элементом*;
- `move(<исходный индекс>, <целевой индекс>)` — перемещает элемент массива данных с местоположения, заданного *исходным индексом*, в местоположение, указанное *целевым индексом*;
- `swap(<индекс 1>, <индекс 2>)` — меняет местами элементы массива данных с указанными *индексами*;
- `pop()` — удаляет из массива данных последний элемент и возвращает его. Если массив пуст, ничего не делает и ничего не возвращает;
- `remove(<индекс>)` — удаляет из массива данных элемент с указанным *индексом* и возвращает этот элемент. Если элемента с заданным *индексом* в массиве не существует, ничего не делает и ничего не возвращает.

Для создания веб-формы, содержащей массив элементов управления, следует использовать функцию, указываемую в качестве содержимого тега `<Formik>` (см. *разд. 16.2*), поскольку для формирования массива элементов понадобится доступ к данным, заносимым в веб-форму и находящимся в объекте хранения.

Вот пример веб-формы, содержащей массив элементов управления и служащей для заполнения массива строк:

```
import { FieldArray } from 'formik';
. . .
<Formik initialValues={{ data: [] }} . . . >
    {formik => (
        <Form>
            <FieldArray name="data">
                {arrayAPI => (
                    <div>
                        {formik.values.data.map((value, index) => (
                            <div key={index}>
                                <Field name={`data.${index}`} />
                                <input type="button" value="Удалить"
                                   onClick={() => {arrayAPI.remove(index)}} />
                            </div>
                        ))}
                        <input type="button" value="Добавить"
                               onClick={() => {arrayAPI.push('')}} />
                    </div>
                )}
            </FieldArray>
            <input type="submit" value="Отправить" />
        </Form>
    )}
</Formik>
```

Функция, выдающая массив элементов управления, также может быть указана в пропе `render` компонента `FieldArray`:

```
<FieldArray name="data"
            render={arrayAPI => (
                . . .
            )}
/>
```

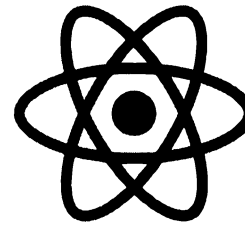
Наконец, можно создать компонент, формирующий сам массив элементов управления, и присвоить его пропу `component` компонента `FieldArray`. В этом случае создаваемому компоненту будут переданы следующие пропы:

- `form` — ссылка на объект построителя веб-форм (описан в *разд. 16.2*);
- `name` — значение одноименного пропа компонента `FieldArray`;
- пропы, хранящие методы для манипуляций с массивом данных (см. ранее): `push`, `unshift` и т. д.

Вот пример компонента с веб-формой, предназначенной для заполнения массива объектов фиксированной структуры, и компонента, реализующего массив элементов управления:

```
<Formik initialValues={{ data: [] }} . . . >
    <Form>
        <h1>Веб-фреймворки</h1>
        <FieldArray name="data" component={MyFieldArray} />
        <input type="submit" value="Отправить" />
    </Form>
</Formik>
. . .
function MyFieldArray(props) {
    return (
        <div>
            {props.form.values.data.map((value, index) => (
                <div key={index}>
                    <Field name={`data.${index}.name`} />
                    <Field as="select" name={`data.${index}.type`}>
                        <option value="client">Клиентский</option>
                        <option value="server">Серверный</option>
                    </Field>
                    <input type="button" value="Удалить"
                           onClick={() => {props.remove(index)}} />
                </div>
            ))}
            <input type="button" value="Добавить"
                onClick={() => {props.push({ name: '', type: 'client' })}} />
        </div>
    );
}
```

ГЛАВА 17



Библиотека Yup: валидация данных

Написание кода, выполняющего валидацию введенных посетителем данных, может стать трудоемкой задачей, особенно если валидация должна выполняться согласно достаточно сложным правилам.

Дополнительная библиотека Yup позволяет описать правила валидации декларативно, в стиле: «это значение должно быть строкой не короче 10 и не длиннее 100 символов» или «это значение должно быть целым числом и укладываться в диапазон от 1 до 10». Кроме того, Yup прекрасно интегрируется с библиотекой Formik, описанной в *главе 16*.

Установка библиотеки Yup производится набором команды:

```
npm install yup
```

> ***На заметку***
>
> Полная документация по библиотеке Yup приведена по интернет-адресу: **https://github.com/jquense/yup**.

17.1. Написание правил валидации. Схемы

Схема Yup — это набор правил, которому должно удовлетворять значение, подвергаемое валидации. Схема представляется особым объектом.

Формирование схемы, содержащей необходимые правила валидации, выполняется следующим образом:

1. Создается пустая схема, описывающая тип данных JavaScript, которому должно принадлежать валидируемое значение, — вызовом одной из функций, объявленных в модуле `yup` и рассматриваемых далее.

   Пример создания пустой схемы, описывающей строковый тип данных:

   ```
   import { string } from 'yup';
   let titleSchema = string();
   ```

2. У полученной пустой схемы указываются дополнительные правила (например, минимальная и максимальная длина строки) — вызовом соответствующих методов этой схемы.

Каждый из таких методов в качестве результата возвращает новую схему, полученную путем добавления к текущей схеме правила, реализуемого вызванным методом. Это позволяет записывать вызовы методов цепочкой.

Пример задания для полученной ранее пустой схемы правила, предписывающего ввод значения максимально допустимой длины в 100 символов:

```
titleSchema = titleSchema.required().max(100);
```

После этого можно выполнить собственно валидацию:

```
// Задано корректное значение
titleSchema.validateSync('Начать разработку приложения');
// Значение проходит валидацию

// Задано некорректное значение: пустая строка
titleSchema.validateSync('');
// Ошибка: ValidationError: this is a required field
```

Каждый из методов, задающих дополнительное правило, способен принимать в качестве одного из параметров выводимое сообщение об ошибке ввода. Оно может быть указано в виде обычной строки:

```
titleSchema = titleSchema.required('Укажите заголовок дела');
titleSchema.validateSync('');
// Ошибка: ValidationError: Укажите заголовок дела
```

В составе сообщения можно вывести одно из служебных значений, предусматриваемых библиотекой Yup. Для этого используется тот же синтаксис, что применяется для вывода результатов каких-либо выражений JavaScript в составе шаблонных строк (однако строка сообщения должна быть обычной, а не шаблонной!).

Вот пример вывода в составе сообщения об ошибке максимально допустимой длины строки:

```
titleSchema = titleSchema.max(100,
              'Заголовок не должен быть длиннее ${max} символов');
titleSchema.validateSync('Это очень очень очень . . . очень ' +
                         'очень длинный заголовок');
// Ошибка: ValidationError: Заголовок не должен быть длиннее 100 символов
```

Сообщение об ошибке также можно указать в виде функции, возвращающей строку с сообщением. В качестве единственного параметра она должна принимать простой объект со следующими свойствами:

- `value` — проверяемое значение в преобразованном виде (например, число, представленное строкой, будет преобразовано в числовой тип);
- `originalValue` — изначальное проверяемое значение до каких бы то ни было преобразований (возвращаясь к ранее приведенному примеру — число в строковом виде);
- свойства, хранящие служебные значения.

Пример:

```
titleSchema = titleSchema.max(100,
              ({ value, max }) =>
                'Заголовок "${value}" не должен быть длиннее ${max} символов');
```

Если сообщение об ошибке ввода не задано, будет выведено стандартное сообщение на английском языке.

Кроме методов, задающих дополнительные правила, объект схемы поддерживает ряд методов, выполняющих всевозможные преобразования. Например, так можно указать Yup удалить у строки начальные и конечные пробелы:

```
titleSchema = string().trim();
```

Некоторые преобразования схема выполняет самостоятельно. Например, числовая схема сама преобразует числа, представленные в виде строк, в числовой тип.

17.1.1. Правила, применимые ко всем типам

Описанные далее методы налагают на схему дополнительные правила и поддерживаются схемами всех типов:

- `required([<`*сообщение*`>])` — валидируемое значение обязательно должно быть указано. Значения `null` и `undefined` считаются некорректными;
- `nullable([<`*сообщение*`>])` — то же самое, что и `required()`, только значение `null` считается корректным;
- `oneOf(<`*массив*`>[, <`*сообщение*`>])` — значение должно совпадать с одним из приведенных в заданном *массиве*:

  ```
  import { string } from 'yup';
  let deedType = string().oneOf(['work', 'home', 'other']);
  ```

  В *сообщении* можно указать служебное значение `values`, которое будет выводиться в виде строки из элементов массива допустимых значений, разделенных запятыми:

  ```
  deedType = string().oneOf(['work', 'home', 'other'],
                    'Тип должен быть значением из списка: ${values}');
  ```

  Значение `undefined` считается корректным, даже если оно не указано в *массиве*. Чтобы исключить его из перечня корректных, следует предварительно вызвать у схемы метод `required()`, например:

  ```
  deedType = string().required().oneOf(['work', 'home', 'other']);
  ```

- `notOneOf(<`*массив*`>[, <`*сообщение*`>])` — значение *не* должно совпадать ни с одним из приведенных в указанном *массиве*.

Также поддерживается преобразующий метод `default()`, задающий *значение по умолчанию*, которое будет использовано, если посетитель не введет значение:

```
default(<значение по умолчанию>|<функция, выдающая значение по умолчанию>)
```

Можно задать *функцию*, не принимающую параметров и возвращающую требуемое значение по умолчанию:

```
import { date } from 'yup';
const deedType2 = string().default('work');
const deedCreatedAt = date().default(() => new Date());
```

17.1.2. Правила валидации строк

Пустая схема, описывающая строковый тип данных, получается вызовом функции `string()`.

Строковая схема поддерживает следующие методы, налагающие на схему дополнительные правила:

- `required([<сообщение>])` — в случае строк не допускаются также пустые строки;
- `length(<длина>[, <сообщение>])` — строка должна иметь строго указанную *длину*. В *сообщении* можно указать служебное значение `length`, хранящее заданную длину;
- `min(<минимальная длина>[, <сообщение>])` — строка должна иметь длину, не менее указанной *минимальной*. В *сообщении* можно указать служебное значение `min`, хранящее заданную минимальную длину;
- `max(<максимальная длина>[, <сообщение>])` — строка должна иметь длину, не более указанной *максимальной*. В *сообщении* можно указать служебное значение `max`, хранящее заданную максимальную длину;
- `email([<сообщение>])` — строка должна представлять собой корректный адрес электронной почты;
- `url([<сообщение>])` — строка должна представлять собой корректный интернет-адрес;
- `uuid([<сообщение>])` — строка должна представлять собой корректный универсальный уникальный идентификатор (UUID);
- `matches()` — строка должна соответствовать заданному *регулярному выражению*, указанному в виде объекта класса `RegExp`. Поддерживаются два формата вызова:

  ```
  matches(<регулярное выражение>[, <сообщение>])
  matches(<регулярное выражение>[, <настройки>])
  ```

  Пример использования первого формата вызова:

  ```
  import { string } from 'yup';
  const postIndexSchema = string().required().matches(/[0-9]{6}/,
                                  'Почтовый индекс должен содержать 6 цифр');
  ```

  Во втором формате вызова *настройки* указываются в виде простого объекта со свойствами:

  - `message` — сообщение об ошибке ввода;
  - `excludeEmptyString` — если `true`, пустые строки будут считаться корректными значениями (разумеется, если у схемы не был вызван метод `required()`, поме-

чающий пустые строки как недопустимые для указания). Если `false`, пустые строки будут считаться некорректными значениями. По умолчанию: `false`.

Пример:

```
const postIndexSchema2 = string().matches(/[0-9]{6}/, {
    message: 'Неверный почтовый индекс',
    excludeEmptyString: true
});
```

- `datetime()` — строка должна представлять собой запись временно́й отметки в формате ISO. Поддерживаются два формата вызова:

```
datetime([<сообщение>])
datetime([<настройки>])
```

Пример использования первого формата:

```
const deedCreatedAt = string().datetime('Это не временная отметка');
```

Во втором формате вызова *настройки* указываются в виде простого объекта со свойствами:

- `message` — сообщение об ошибке ввода;
- `allowOffset` — если `true`, в составе заданной временно́й отметки допускается наличие временно́й зоны. Если `false`, обозначение временно́й зоны не допускается, и время считается принадлежащим временно́й зоне UTC. По умолчанию: `false`;
- `precision` — допустимое количество цифр в дробной части количества секунд. Если `null`, секунды должны быть выражены целым числом. По умолчанию: `null`.

Пример:

```
const deedCreatedAt2 = string().datetime({
    message: 'Это не временная отметка',
    allowOffset: true,
    precision: 3
});
```

Строковая схема также поддерживает ряд методов, выполняющих различные преобразования:

- `trim()` — удаляет из строки все начальные и конечные пробелы;
- `lowercase()` — приводит символы в строке к нижнему регистру;
- `uppercase()` — приводит символы в строке к верхнему регистру;
- `ensure()` — преобразует значения `null` и `undefined` в пустые строки.

17.1.3. Правила валидации чисел

Пустая схема, описывающая числовой тип данных, получается вызовом функции `number()`.

Числовая схема позволяет валидировать как значения собственно числового типа, так и представленные в виде строк. Последние она преобразует в числовой тип, используя функцию JavaScript `parseFloat()` и предварительно удалив из строки все начальные и конечные пробельные символы.

Числовая схема поддерживает следующие методы, налагающие дополнительные правила:

- `integer([<`*сообщение*`>])` — число должно быть целым;
- `min(<`*минимум*`>[, <`*сообщение*`>])` — число должно быть больше или равно указанному *минимуму*. В *сообщении* можно указать служебное значение `min`, хранящее заданный минимум;
- `moreThan(<`*минимум*`>[, <`*сообщение*`>])` — число должно быть строго больше указанного *минимума*. В *сообщении* можно указать служебное значение `more`, хранящее заданный минимум;
- `max(<`*максимум*`>[, <`*сообщение*`>])` — число должно быть меньше или равно указанному *максимуму*. В *сообщении* можно указать служебное значение `max`, хранящее заданный максимум;
- `lessThan(<`*максимум*`>[, <`*сообщение*`>])` — число должно быть строго меньше указанного *максимума*. В *сообщении* можно указать служебное значение `less`, хранящее заданный максимум;
- `positive([<`*сообщение*`>])` — число должно быть положительным;
- `negative([<`*сообщение*`>])` — число должно быть отрицательным.

Преобразующие методы, поддерживаемые числовой схемой:

- `truncate()` — преобразует число в целочисленное, удаляя все знаки после точки и саму точку;
- `round([<`*функция*`>='round'])` — округляет число до целого с применением указанной функции JavaScript. Поддерживаются функции `'floor'`, `'ceil'`, `'trunc'` и `'round'`.

17.1.4. Правила валидации логических величин

Пустая схема, описывающая логический тип данных, получается вызовом функции `boolean()`. Схема валидирует как значения логического типа, так и представленные в виде чисел и строк. При этом строки `'true'` и `'1'`, а также числа 1 преобразуются в значения `true`, а строки `'false'`, `'0'` и числа 0 — в `false`. Никаких специфических методов эта схема не поддерживает.

17.1.5. Правила валидации временны́х отметок

Пустая схема, описывающая временну́ю отметку, получается вызовом функции `date()`. Она позволяет валидировать временны́е отметки, представленные в виде объектов класса `Date` или строк в формате ISO. Последние она преобразует в объекты класса `Date`, применяя собственную функцию.

Методы, налагающие дополнительные условия:

- `min(<минимум>[, <сообщение>])` — временна́я отметка должна быть больше или равной указанному *минимуму*. *Минимум* может быть задан в виде объекта класса `Date` или строки в формате ISO. В *сообщении* можно указать служебное значение `min`, хранящее заданный минимум;
- `max(<максимум>[, <сообщение>])` — временна́я отметка должна быть меньше или равной указанному *максимуму*. *Максимум* может быть задан в виде объекта класса `Date` или строки в формате ISO. В *сообщении* можно указать служебное значение `max`, хранящее заданный максимум.

17.1.6. Правила валидации массивов

Пустую схему, описывающую массив, можно получить, вызвав функцию `array([<схема элементов>])`.

В вызове функции `array()` допускается указать *схему элементов*, которой должны соответствовать все элементы массива. Вот пример схемы массива, который должен содержать лишь целые числа от 1 до 5:

```
import { array, number } from 'yup';
let scoreScheme = array(number().integer().min(1).max(5));
```

Поддерживаемые методы, налагающие дополнительные условия:

- `of(<схема>)` — указывает *схему*, которой должны соответствовать элементы массива. Аналогичен вызову функции `array()` с параметром. Пример:

  ```
  scoreScheme = array().of(number().integer().min(1).max(5));
  ```

- `length(<размер>[, <сообщение>])` — массив должен иметь строго указанный *размер*. В *сообщении* можно указать служебное значение `length`, хранящее заданный размер;
- `min(<минимальный размер>[, <сообщение>])` — массив должен иметь размер больший или равный указанному *минимальному*. В *сообщении* можно указать служебное значение `min`, хранящее заданный минимальный размер;
- `max(<максимальный размер>[, <сообщение>])` — массив должен иметь размер меньший или равный указанному *максимальному*. В *сообщении* можно указать служебное значение `max`, хранящее заданный максимальный размер.

Поддерживаемые преобразующие методы:

- `ensure()` — преобразует:
  - значение `null` или `undefined` — в пустой массив;

- любое значащее значение, не являющееся массивом, — в массив из одного элемента, содержащего заданное значение;
- массив — возвращает без преобразования;

- `compact(<исключающая функция>)` — удаляет из массива элементы, в отношении которых заданная *исключающая функция* вернет значение `false`. *Исключающая функция* в качестве параметра должна принимать очередной элемент валидируемого массива. Пример кода, удаляющего из массива отрицательные числа:

```
const posValuesScheme = array().compact(el => el >= 0);
```

- `json()` — пытается преобразовать строковое представление массива (наподобие: `'[1, 2, 3]'`) в массив вызовом статического метода `parse()` класса `JSON`. Если попытка не увенчается успехом, возбуждается исключение типа `TypeError`. Пример:

```
const numbers = array(number()).json();
```

17.1.7. Правила валидации объектов

Пустая схема, описывающая объект, получается с помощью функции `object([<схема структуры>])`.

В вызове функции `object()` можно указать *схему структуры*, описывающую набор свойств валидируемых объектов. *Схема структуры* записывается в виде простого объекта, содержащего свойства, которые обязательно должны или могут присутствовать в валидируемых объектах. Значениями этих свойств должны быть схемы, описывающие правила, которым должны соответствовать значения соответствующих свойств валидируемых объектов. У схем, описывающих значения свойств, которые обязательно должны присутствовать в объектах, следует вызвать метод `required()` (см. *разд. 17.1.1*). У необязательных свойств этого делать не нужно. Кроме того, у схем, описывающих значения свойств, можно указать значения по умолчанию, применив метод `default()` (см. *разд. 17.1.1*).

Вот пример схемы, описывающей объект с обязательными свойствами `name1`, `name2`, `subscribe` (значение по умолчанию: `true`) и необязательным свойством `age` (допускаются только целочисленные положительные значения):

```
import { object, string, boolean, number } from 'yup';
let userScheme = object({
    name1: string().required(),
    name2: string().required(),
    subscribe: boolean().default(true),
    age: number().integer().positive()
});
```

Схемой поддерживается метод `shape(<схема структуры>)`, задающий у объектов *схему структуры*. Он аналогичен вызову функции `object()` с параметром. Пример:

```
userScheme = object().shape( . . . );
```

Последующий вызов метода `shape()` добавляет заданные в нем свойства к схеме структуры, указанной в предыдущем вызове этого же метода (или функции `object()`). Если в обоих схемах структуры присутствует одно и то же свойство, его схема, указанная в последующем вызове метода `shape()`, перекроет схему из предыдущего вызова. Следующий пример создает ту же схему, что и приведенный ранее:

```
userScheme = object({
    name1: string().required(),
    name2: number()
}).shape({
    name2: string().required(),
    subscribe: boolean().default(true)
}).shape({
    age: number().integer().positive()
});
```

Может оказаться полезным метод `when()`, накладывающий на текущее свойство проверяемого объекта дополнительное условие в зависимости от значения другого свойства, называемого *проверочным*, или значений нескольких проверочных свойств, приведенных в указанном *массиве*:

```
when(<имя проверочного свойства>|<массив имен проверочных свойств>,
                                    <настройки>)
```

Имя проверочного свойства задается в виде строки. Имена отдельных проверочных свойств в заданном *массиве* также записываются в виде строк.

Настройки должны быть заданы в виде простого объекта со свойствами:

- `is` — функция, принимающая в качестве параметров значения проверочных свойств, указанных в первом параметре метода `when()`, и возвращающая значение `true` или `false`;
- `then` — условие, накладываемое на текущее свойство проверяемого объекта, если функция из свойства `is` вернет `true`.

  Значением свойства должна быть функция, в качестве единственного параметра получающая предварительно настроенную схему, которая описывает текущее свойство проверяемого объекта. В теле этой функции на схему следует наложить необходимые дополнительные условия и вернуть эту схему в качестве результата;
- `otherwise` — условие, накладываемое на текущее свойство проверяемого объекта, если функция из свойства `is` вернет `false`. В остальном аналогично свойству `then`.

Пример схемы, требующей, чтобы в свойство `desc` обязательно была занесена строка длиной не менее 20 символов, если свойство `type` хранит значение `'work'`:

```
const deedSchema = object({
    title: string().required(),
```

```
    type: string().required().default('home'),
    desc: string().when('type', {
        is: type => type == 'work',
        then: scheme => scheme.required().min(20),
        otherwise: scheme => scheme
    })
});
```

Пример схемы, накладывающей на свойство `desc` такое же условие, если свойство `type` хранит значение `'work'`, или в свойство `title` занесена строка длиной более 10 символов:

```
const deedSchema2 = object({
    title: string().required(),
    type: string().required().default('home'),
    desc: string().when(['title', 'type'], {
        is: (title, type) => (title.length > 10 || type == 'work'),
        then: scheme => scheme.required().min(20),
        otherwise: scheme => scheme
    })
});
```

Преобразующий метод `json()` пытается преобразовать строковое представление объекта (наподобие: `'{"name1": "Ivan", "name2": "Ivanov"}'`) в объект вызовом статического метода `parse()` класса `JSON`. Если попытка не увенчается успехом, возбуждается исключение типа `TypeError`.

17.1.8. Написание своих дополнительных условий и преобразований

Библиотека Yup также позволяет добавить в схему свои собственные дополнительные условия и преобразования.

Дополнительное условие добавляется в схему вызовом метода `test()` схемы:

`test(`*<имя условия>*`, `*<сообщение об ошибке>*`, `*<функция проверки>*`)`

Имя условия задается в виде строки и должно быть уникальным в пределах приложения. Оно используется внутренними механизмами Yup.

Сообщение об ошибке может быть как строкой, так и функцией (подробности — в *разд. 17.1*).

Функция проверки должна принимать два параметра: само проверяемое значение в преобразованном виде и простой объект с дополнительными сведениями об этом значении. Наибольший интерес представляет свойство `originalValue` этого объекта, хранящее проверяемое значение в изначальном виде.

Функция проверки должна вернуть `true`, если проверяемое значение корректно, или `false` — в противном случае. Если *функция* является асинхронной, она также может вернуть промис, содержащий в качестве нагрузки значение `true` или `false`.

Вот пример условия, требующего, чтобы возраст посетителя находился в диапазоне от 18 до 65:

```
import { number } from 'yup';
const ageScheme = number().required().test(
    'workAge',
    'Посетитель должен быть трудоспособного возраста',
    val => (val >= 18 && val < 65)
);
```

Метод `test()` возвращает ссылку на текущую схему, поэтому его вызовы можно записывать цепочкой. Допускается произвольное количество вызовов этого метода.

Для добавления в схему своего преобразования следует использовать метод `transform()` схемы:

```
transform(<преобразующая функция>)
```

Указываемая *преобразующая функция* должна принимать два параметра: значение, подлежащее преобразованию, в предварительно преобразованном виде, и то же значение, только изначальное. Возвращать она должна значение, подвергнутое преобразованию.

Вот пример схемы, округляющей цену до копеек:

```
const priceScheme = number().required().transform(
    val => Math.round(val * 100) / 100
);
```

Метод `transform()` также возвращает ссылку на текущую схему, поэтому его вызовы можно записывать цепочкой. Допускается произвольное количество вызовов этого метода.

17.2. Валидация данных

17.2.1. Использование только инструментов Yup

Для непосредственно валидации значений применяются следующие методы схемы:

- `validate(<значение>)` — валидирует заданное *значение* асинхронно. Возвращает промис, который:
  - подтверждается — если *значение* корректно.

    В качестве нагрузки промис хранит валидируемое значение, подвергнутое преобразованиям, которые были указаны в схеме (например, число, записанное в строковом виде, будет преобразовано в числовой тип);
  - отклоняется — если *значение* некорректно.

    В качестве нагрузки промис будет хранить объект класса `ValidationError` из модуля `yup`, описывающий выявленную ошибку ввода. Свойство `errors` этого класса хранит массив с сообщениями о выявленных ошибках.

Пример:

```
import { string } from 'yup';
const titleSchema = string().required('Укажите заголовок').trim();
. . .
titleSchema.validate(title).then(val => {
    setTitle(val);
}).catch(err => {
    setTitleError(err.errors[0]);
});
```

- `validateSync(<значение>)` — валидирует заданное *значение* синхронно. Возвращает преобразованное значение, если оно корректно. Если же значение некорректно, возбуждает исключение типа `ValidationError`. Пример:

```
try {
    setTitle(titleSchema.validateSync(title));
}
catch (err) {
    setTitleError(err.errors[0]);
}
```

- `isValid(<значение>)` — валидирует заданное *значение* асинхронно. Возвращает всегда подтверждаемый промис, в качестве нагрузки хранящий значение `true`, если *значение* корректно, и `false` — в противном случае. Пример:

```
if (await titleSchema.isValid(title))
    // Все в порядке: заголовок дела указан
```

- `isValidSync(<значение>)` — валидирует заданное *значение* синхронно. Возвращает значение `true`, если *значение* корректно, и `false` — в противном случае. Пример:

```
if (titleSchema.isValidSync(title))
    // Все в порядке: заголовок дела указан
```

- `cast(<значение>)` — пытается выполнить преобразование указанного *значения* в вид, задаваемый текущей схемой (например, числа, записанного в виде строки, в числовой тип). Возвращает преобразованное значение, если попытка удалась. В противном случае может вернуть `null`, `NaN` или произвольную строку. Пример:

```
import { number } from 'yup';
const ageScheme = number().integer();
. . .
const ageProcessed = ageScheme.cast(age);
if (typeof ageProcessed === 'number' && !ageProcessed.isNaN())
    // Все в порядке: возраст указан корректно
```

17.2.2. Использование Yup совместно с Formik

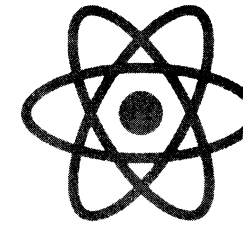
Если веб-форма, в которую заносятся валидируемые данные, построена с применением библиотеки Formik (см. *главу 16*), задача валидации данных существенно упрощается.

Компонент `Formik`, реализующий построитель веб-форм, поддерживает проп `validationSchema`. Ему присваивается объектная схема Yup (см. *разд. 17.1.7*), описывающая объект со свойствами, имена которых совпадают с наименованиями элементов управления и которые хранят схемы, указывающие правила валидации значений, заносимых в эти элементы.

Вот пример схемы, описывающей валидацию значений, которые заносятся в веб-форму добавления нового дела:

```
import { object, string } from 'yup';
. . .
<Formik initialValues={{ title: '', desc: '', image: '' }}
        . . .
        validationSchema={object({
            title: string().required('Укажите заголовок дела')
                           .max(100, 'Заголовок дела не должен быть ' +
                                     'длиннее ${max} символов')
                           .trim(),
            desc: string().trim(),
            image: string()
        })}
>
    . . .
<Formik>
```

ГЛАВА 18



Библиотека ReactFire: интеграция с Google Firebase

Дополнительная библиотека ReactFire является надстройкой над Google Firebase SDK и несколько упрощает работу с серверными приложениями Firebase (см. *главу 2*). Библиотека устанавливается подачей команды:

```
npm install reactfire
```

> ***НА ЗАМЕТКУ***
>
> Полное описание библиотеки ReactFire приведено на странице **https://github.com/FirebaseExtended/reactfire**.

Все компоненты и хуки ReactFire, описываемые далее, объявлены в модуле `reactfire`.

Можно сказать, что веб-приложение, работающее с Firebase посредством ReactFire, состоит из трех уровней. Эти уровни и их реализация рассмотрены далее.

18.1. Первый уровень: провайдер серверного веб-приложения

На первом, самом «нижнем» уровне веб-приложения, написанного с использованием ReactFire, располагается *провайдер серверного веб-приложения*. Он предоставляет компонентам-потомкам доступ к объекту, представляющему серверное приложение.

Провайдер серверного приложения представляется компонентом `FirebaseAppProvider`. В пропе `firebaseConfig` этого компонента указывается объект, содержащий параметры подключения к серверному приложению Firebase (этот объект был описан в *разд. 2.1*).

Все компоненты клиентского React-приложения, которым требуется работать с серверным приложением Google Firebase, должны являться потомками этого провайдера.

Как правило, провайдер серверного приложения помещают в состав содержимого, выводимого стартовым модулем, а его потомком делают компонент, содержащий два других ключевых провайдера (описаны в *разд. 18.2*).

Пример:

```
import { FirebaseAppProvider } from 'reactfire';
import firebaseConfig from './firebase.js';
import Providers from './Providers.js';

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(
    <React.StrictMode>
        <FirebaseAppProvider firebaseConfig={firebaseConfig}>
            <Providers />
        </FirebaseAppProvider>
    </React.StrictMode>
);
```

18.2. Второй уровень: провайдеры базы данных и подсистемы разграничения доступа

На втором уровне находятся два вложенных друг в друга провайдера, в свою очередь, вкладываемые в провайдер серверного приложения:

- *базы данных* — предоставляет компонентам-потомкам доступ к базе данных.

  Провайдер базы данных представляется компонентом `FirestoreProvider`. В пропе `sdk` этого компонента указывается объект, представляющий базу данных. Этот объект можно получить, вызвав функцию `getDatabase()` из модуля `firebase/database` (описана в *разд. 2.4*), передав ей в качестве параметра объект серверного приложения.

  Объект серверного приложения можно получить вызовом хука `useFirebaseApp()`, который получает объект от провайдера, описанного в *разд. 18.1*;

- *подсистемы разграничения доступа* — предоставляет компонентам-потомкам доступ к упомянутой подсистеме.

  Этот провайдер представляется компонентом `AuthProvider`. В его пропе `sdk` указывается объект, представляющий подсистему разграничения доступа. Получить его можно вызовом функции `getAuth()` из модуля `firebase/auth` (см. *разд. 2.2*), передав ей всё тот же объект серверного приложения.

Порядок вложенности этих провайдеров друг в друга может быть произвольным.

Провайдеры базы данных и подсистемы разграничения доступа обычно размещаются в отдельном компоненте. А в самих провайдерах помещается компонент приложения или провайдер маршрутизации React Router.

В листинге 18.1 приведен код компонента `Providers`, содержащего провайдеры базы данных, подсистемы разграничения доступа и маршрутизации.

Листинг 18.1. Код компонента `Providers`, содержащего провайдеры базы данных, подсистемы разграничения доступа и маршрутизации

```
import { getDatabase } from 'firebase/database';
import { getAuth } from 'firebase/auth';
import { FirestoreProvider, AuthProvider, useFirebaseApp }
       from 'reactfire';
import { RouterProvider } from 'react-router-dom';
import router from './routes.js';

export default function Providers() {
    const firebase = useFirebaseApp();
    const database = getDatabase(firebase);
    const auth = getAuth(firebase);

    return (
        <FirestoreProvider sdk={database}>
            <AuthProvider sdk={auth}>
                <RouterProvider router={router} />
            </AuthProvider>
        </FirestoreProvider>
    );
}
```

18.3. Третий уровень: компоненты-потребители

На третьем, наиболее «верхнем» уровне находятся компоненты-потребители, использующие базу данных и подсистему разграничения доступа. Для доступа к ним компоненты-потребители используют особые хуки.

18.3.1. Получение сведений о текущем посетителе

Получить сведения о текущем посетителе у подсистемы разграничения доступа можно посредством следующих хуков:

- `useSigninCheck()` — позволяет выяснить, выполнил ли какой-либо посетитель вход в приложение, и получить сведения об этом посетителе. Одновременно подписывает компонент на отслеживание изменений в статусе посетителя: входа в приложение и выхода из него.

  В качестве результата хук возвращает объект класса `ObservableStatus`, который представляет значение, получаемое от бэкенда (в рассматриваемом случае — объект со сведениями о статусе посетителя), позволяет отследить статус получения этого значения и в случае неудачи узнать, какая ошибка не позволила получить его. Этот класс поддерживает следующие основные свойства:

- `status` — статус получения значения в виде одной из строк:
  - `'loading'` — идет получение значения;
  - `'success'` — значение успешно получено;
  - `'error'` — при получении значения возникла ошибка;
- `data` — само значение, получаемое от бэкенда, или `undefined`, если оно еще не получено или его получить не удалось;
- `error` — возникшая при получении значения ошибка, представленная объектом класса `Error` или какого-либо производного от него класса, или `undefined`, если значение было успешно получено.

Свойство `data` объекта, возвращенного хуком `useSigninCheck()`, будет хранить объект класса `SigninCheckResult`, который представляет сведения о статусе текущего посетителя. Вот наиболее полезные свойства этого класса:

- `signedIn` — `true`, если какой-либо посетитель в текущий момент выполнил вход в приложение, `false` — в противном случае;
- `user` — объект со сведениями о текущем посетителе или `null`, если вход не был выполнен;
- `errors` — простой объект, свойства которого хранят ошибки, возникшие при попытке определить статус посетителя, или «пустой» объект, если статус посетителя был получен без проблем.

Вот пример использования этого хука:

```
import { useSigninCheck } from 'reactfire';
export default function App() {
    const { status, data: check } = useSigninCheck();
    const user = (status == 'success' && check.signedIn) ?
                                                check.user : null;
    . . .
    return (
        . . .
        <p>{user ? user.email : 'Todos'}</p>
        . . .
    );
}
```

- `useUser()` — извлекает сведения о текущем посетителе, одновременно подписывая компонент на отслеживание изменений этих сведений (например, в результате входа или выхода).

  Возвращает объект класса `ObservableStatus` (описан ранее), свойство `data` которого будет хранить объект со сведениями о текущем посетителе, или `null`, если ни один посетитель еще не выполнил вход.

  Пример:

```
import { useUser } from 'reactfire';
export default function TodoList() {
    const { status, data: user } = useUser();
```

```
    if (status == 'loading')
        return null;
    else
        return <__TodoList user={user} />;
}
```

Эти хуки получают объект подсистемы разграничения доступа, необходимый им для работы, от провайдера `AuthProvider` (см. *разд. 18.2*). Естественно, что этот провайдер обязательно должен присутствовать в одном из родительских компонентов.

18.3.2. Извлечение данных из базы

Для извлечения данных из базы применяются три следующих хука:

- `useDatabase()` — возвращает объект базы данных, используемый для создания ссылок на пути к коллекциям и отдельным документам.

  Хук получает объект подсистемы разграничения доступа, необходимый ему для работы, от провайдера `DatabaseProvider` (см. *разд. 18.2*). Естественно, что этот провайдер обязательно должен присутствовать в одном из родительских компонентов;

- `useDatabaseListData()` — извлекает все документы из коллекции, расположенной по указанной *ссылке* или описанной заданным *запросом*:

  `useDatabaseListData(<`*ссылка на путь*`>|<`*запрос*`>[, <`*настройки*`>={}])`

  Ссылку на путь можно получить вызовом функции `ref()` из модуля `firebase/database`, *запрос* — вызовом функции `query()` из того же модуля (обе функции описаны в *разд. 2.4*). Объект базы данных, необходимый для создания ссылки на путь или запроса, можно получить вызовом хука `useDatabase()`.

  Настройки задаются в виде простого объекта. Наиболее полезной является настройка `idField`, задающая имя атрибута, который будет создаваться в каждом из извлеченных документов и сохранит уникальный идентификатор этого документа. Если эта настройка не указана, атрибут с идентификатором получит имя `NO_ID_FIELD`.

  Хук возвращает объект класса `ObservableStatus` (см. *разд. 18.3.1*), свойство `data` которого хранит массив с извлеченными из коллекции документами, или `null`, если коллекция по заданному пути пуста или не существует.

  Пример:

```
import { useFirestore, useDatabaseListData } from 'reactfire';
import { ref } from 'firebase/database';

function __TodoList({ user }) {
    const firestore = useFirestore();
    const r = ref(firestore, `users/${user.uid}/todos`);
    const { status, data: list } = useDatabaseListData(r, {
        idField: 'key',
    });
    . . .
```

```
    if (status == 'loading')
        return null;
    else
        return (
            <table>
                {list.map(item => (
                    . . .
                ))}
            </table>
        );
}
```

- `useDatabaseObjectData()` — **извлекает документ, расположенный по указанной** *ссылке*:

  `useDatabaseObjectData(<`*ссылка на путь*`>[, <`*настройки*`>={}])`

  Ссылка на путь **и** *настройки* **указываются в том же формате, что и у хука** `useDatabaseListData()`.

  Хук **возвращает объект класса** `ObservableStatus` **(см.** *разд. 18.3.1*), **свойство** `data` **которого хранит извлеченный документ, или** `null`, **если документ по заданному пути не существует.**

  Пример:

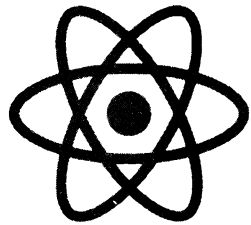
```
import { useParams } from 'react-router-dom';
import { useFirestore, useDatabaseObjectData } from 'reactfire';
import { ref } from 'firebase/database';

function __TodoDetail({ user }) {
    const { key } = useParams();

    const firestore = useFirestore();
    const r = ref(firestore, `users/${user.uid}/todos/${key}`);
    const { status, data: todo} = useDatabaseObjectData(r);

    if (status == 'loading')
        return null;
    else
        return (
            <section>
                <h1>{todo.title}</h1>
                . . .
            </section>
        );
}
```

ГЛАВА 19



Библиотека React Awesome Reveal: анимация

Дополнительная библиотека React Awesome Reveal позволяет реализовывать у элементов страницы и целых компонентов разнообразные анимационные эффекты: «проявление», «выезд» со стороны, увеличение, «пульсацию», покачивание и пр. Установить ее можно, подав команду:

```
npm install react-awesome-reveal
```

> ***НА ЗАМЕТКУ***
>
> Полное описание библиотеки React Awesome Reveal находится по интернет-адресу: **https://react-awesome-reveal.morello.dev/**.

Все описываемые далее компоненты объявлены в модуле `react-awesome-reveal`.

19.1. Анимационные эффекты

Анимационные эффекты воспроизводятся дважды: когда анимируемый элемент страницы или компонент появляется на странице (прямая анимация) и когда он исчезает (обратная).

19.1.1. Компоненты, реализующие анимационные эффекты

Для реализации анимационных эффектов используется ряд компонентов. Элементы страницы или компоненты, подвергаемые анимации, указываются в качестве потомка соответствующего анимирующего компонента.

Компоненты, реализующие анимационные эффекты:

- `Fade` — постепенное «проявление» анимируемого содержимого, возможно, с одновременным «выездом» с указанной стороны:

```
import { Fade } from 'react-awesome-reveal';
. . .
```

```
<Fade>
    <h1>Перечень дел</h1>
</Fade>
```

Пример анимирования сразу нескольких элементов:

```
<Fade>
    <h1>{todo.title}</h1>
    <p>{todo.createdAt}</p>
    <p>{todo.desc}</p>
</Fade>
```

Проп `direction` задает сторону, из которой будет «выезжать» анимируемое содержимое. Он поддерживает следующие значения: `"down"` (сверху вниз), `"left"` (слева), `"right"` (справа), `"up"` (снизу вверх), `"top-left"` (из левого верхнего угла), `"top-right"` (из правого верхнего угла), `"bottom-left"` (из левого нижнего угла), `"bottom-right"` (из правого нижнего угла) и `undefined` («проявление» по месту). Значение по умолчанию: `undefined`. Пример:

```
<Fade direction="left">
    <h1>Перечень дел</h1>
</Fade>
```

- `Flip` — поворот вокруг горизонтальной или вертикальной оси с постепенным «проявлением».

  Проп `direction` указывает координатную ось, вокруг которой будет производиться поворот, и угол поворота:

  - `"horizontal"` — поворот на 90° вокруг горизонтальной оси;
  - `"vertical"` — поворот на 90° вокруг вертикальной оси;
  - `undefined` — поворот на 360° вокруг вертикальной оси;

- `Rotate` — поворот вокруг оси Z с постепенным «проявлением».

  Проп `direction` обозначает точку анимируемого содержимого, вокруг которой будет производиться поворот, угол и направление поворота:

  - `"top-left"` — поворот на 90° вокруг левого нижнего угла анимируемого содержимого против часовой стрелки;
  - `"top-right"` — поворот на 90° вокруг правого нижнего угла анимируемого содержимого по часовой стрелке;
  - `"bottom-left"` — поворот на 90° вокруг левого нижнего угла анимируемого содержимого по часовой стрелке;
  - `"bottom-right"` — поворот на 90° вокруг правого нижнего угла анимируемого содержимого против часовой стрелки.
  - `undefined` — поворот на 180° вокруг правого нижнего угла анимируемого содержимого против часовой стрелки.

  Значение по умолчанию: `undefined`.

Пример:

```
import { Rotate } from 'react-awesome-reveal';
. . .
<Rotate direction="bottom-left">
    <h1>Перечень дел</h1>
</Rotate>
```

- `Zoom` — постепенное увеличение с возможным «выскакиванием» с заданной стороны.

  Сторону, с которой «выскочит» элемент, задает проп `direction`: `"down"` (сверху), `"left"` (слева), `"right"` (справа), `"up"` (снизу) и `undefined` (увеличение на месте). Значение по умолчанию: `undefined`;

- `Bounce` — «пульсация», возможно, с «выездом» с указанной стороны.

  Сторону задает проп `direction`, поддерживающий значения: `"down"` (сверху), `"left"` (слева), `"right"` (справа), `"up"` (снизу) и `undefined` («пульсация» на месте). Значение по умолчанию: `undefined`;

- `Slide` — «выезд» с указанной стороны.

  Сторона, откуда будет производиться «выезд», задается пропом `direction`: `"down"` (сверху), `"left"` (слева), `"right"` (справа), `"up"` (снизу) и `undefined` (то же самое, что и `"left"`). Значение по умолчанию: `undefined`. Пример:

```
import { Slide } from 'react-awesome-reveal';
. . .
<Slide direction="top">
    <h1>Перечень дел</h1>
</Slide>
```

- `Roll` — «вылет» с левой стороны с поворотом на небольшой угол и постепенным «проявлением»;
- `JackInTheBox` — постепенное увеличение с покачиванием вокруг оси Z;
- `Hinge` — анимируемое содержимое как будто повисает на одном гвозде, покачиваясь, после чего падает.

Анимационный эффект будет воспроизводиться только в том случае, если анимируемое содержимое находится в области видимости (т. е. видимо посетителю). Это сделано для повышения производительности.

19.1.2. Общие параметры анимационных эффектов

Все компоненты, реализующие анимационные эффекты, поддерживают ряд параметров, задаваемых следующими пропами:

- `duration` — продолжительность анимации в виде целого числа в миллисекундах (по умолчанию: 1000 мс, или 1 секунда):

```
import { Fade } from 'react-awesome-reveal';
. . .
```

```
<Fade duration={500}>
    <h1>Перечень дел</h1>
</Fade>
```

- `delay` — длительность задержки перед началом анимации в виде целого числа в миллисекундах (по умолчанию: `0`).

  В следующем примере заголовок **Перечень дел** появится на странице сразу же, а расположенная ниже его таблица с перечнем дел — спустя 2 секунды:

```
<Fade>
    <h1>Перечень дел</h1>
</Fade>
<Fade delay={2000}>
    <table>
        . . .
    </table>
</Fade>
```

- `fraction` — доля анимируемого содержимого, которая должна присутствовать в области видимости, чтобы это содержимое было анимировано. Указывается в виде вещественного числа от `0.0` (содержимое будет анимироваться сразу же, как только хотя бы один его пиксел покажется в области видимости) до `1.0` (содержимое должно быть видимо полностью). Значение по умолчанию: `0.0`;
- `triggerOnce` — если `false`, элемент будет анимироваться каждый раз, когда он входит в область видимости или выходит из нее. Если `true`, элемент будет анимирован лишь один раз — когда он впервые попадет в область видимости. По умолчанию: `false`;
- `className` — имена стилевых классов, которые требуется указать у самого анимирующего компонента. По умолчанию: `undefined` (отсутствие дополнительных стилевых классов);
- `childClassName` — имена стилевых классов, которые требуется указать у анимируемых элементов. По умолчанию: `undefined` (отсутствие дополнительных стилевых классов);
- `style` — встроенный стиль, задаваемый у самого анимирующего компонента. По умолчанию: `undefined` (отсутствие встроенного стиля);
- `childStyle` — встроенный стиль, задаваемый у анимируемых элементов. По умолчанию: `undefined` (отсутствие встроенного стиля);
- `onVisibilityChange` — функция, вызываемая каждый раз, когда анимируемое содержимое входит в область видимости или выходит из нее. Должна принимать два параметра:
  - признак положения анимируемого содержимого в области видимости — `true` (находится в области видимости) или `false` (находится вне области видимости);
  - элемент анимируемого содержимого в виде объекта встроенного класса `IntersectionObserverEntry`.

Если в составе анимируемого содержимого находится более одного элемента, заданная функция будет вызвана для каждого из этих элементов.

Результата заданная функция возвращать не должна.

19.1.3. Анимирование нескольких элементов (компонентов)

В анимирующий компонент можно поместить произвольное количество элементов страницы и (или) компонентов — и все эти элементы (компоненты) будут анимированы.

По умолчанию анимационные эффекты, реализуемые анимирующим компонентом, у всех этих элементов (компонентов) будут воспроизводиться одновременно. Например, следующие три абзаца одновременно «проявятся» на странице:

```
import { Fade } from 'react-awesome-reveal';
. . .
<Fade>
    <p>Абзац №1</p>
    <p>Абзац №2</p>
    <p>Абзац №3</p>
</Fade>
```

Если же требуется, чтобы анимационные эффекты у анимируемых элементов (компонентов) воспроизводились последовательно, один за другим, следует указать у анимирующего компонента проп `cascade` со значением `true`. Например, следующие три абзаца «проявятся» на странице последовательно, друг за другом:

```
<Fade cascade>
    <p>Абзац №1</p>
    <p>Абзац №2</p>
    <p>Абзац №3</p>
</Fade>
```

Можно указать задержку между окончанием воспроизведения анимации предыдущего анимируемого элемента и началом воспроизведения анимации следующего. Задержка указывается относительно продолжительности анимации и задается в пропе `damping` в виде вещественного числа. Число `1.0` обозначает задержку, равную продолжительности анимации, числа от `0.0` до `1.0` — меньшую продолжительности анимации, а числа больше `1.0` — бóльшую. Значение пропа по умолчанию: `0.5`. Пример:

```
<Fade cascade damping={0.75}>
    <p>Абзац №1</p>
    <p>Абзац №2</p>
    <p>Абзац №3</p>
</Fade>
```

19.1.3.1. Анимирование символов текста

Чтобы анимировать отдельные символы текста, следует заключить нужный фрагмент текста в анимирующий компонент и задать у последнего проп `cascade` со значением `true`:

```
import { Fade } from 'react-awesome-reveal';
. . .
<h1><Fade cascade>Создание нового дела</Fade></h1>
```

19.2. Специальные эффекты

Специальные эффекты воспроизводятся при монтировании или обновлении компонентов. Обычно они используются, чтобы привлечь внимание посетителя к какому-либо элементу страницы или компоненту.

Специальный эффект реализуется компонентом `AttentionSeeker`. Разновидность эффекта указывается в его пропе `effect` в виде одного из следующих значений:

- `"bounce"` — подпрыгивание:

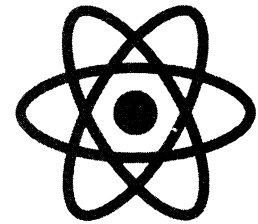
```
import { AttentionSeeker } from 'react-awesome-reveal';
. . .
<input onChange={evt => {setTitle(evt.target.value)} } />
{!title && (
    <AttentionSeeker effect="bounce">
        <p>Введите заголовок</p>
    </AttentionSeeker>
)}
```

- `"flash"` — мерцание;
- `"headShake"` — небольшая «встряска» в горизонтальном направлении;
- `"heartBeat"` — пульсация;
- `"jello"` — покачивание с растягиванием;
- `"pulse"` — медленное однократное увеличение и возвращение к изначальным размерам;
- `"rubberBand"` — резкое однократное растягивание по горизонтали и возвращение к изначальным размерам;
- `"shake"` или `"shakeX"` — интенсивная «встряска» в горизонтальном направлении;
- `"shakeY"`— интенсивная «встряска» в вертикальном направлении;
- `"swing"` — простое покачивание;
- `"tada"` — резкое увеличение в размерах с одновременным покачиванием;
- `"wobble"` — интенсивное покачивание по горизонтальной дуге.

Компонент `AttentionSeeker` также поддерживает пропы `duration`, `delay`, `fraction`, `triggerOnce`, `className`, `childClassName`, `style`, `childStyle`, `onVisibilityChange`, `cascade` и `damping`, описанные в *разд. 19.1.2* и *19.1.3*. Пример:

```
import { AttentionSeeker } from 'react-awesome-reveal';
. . .
<AttentionSeeker effect="wobble" duration={5000} cascade>
    <p>Долго мне еще трястись?</p>
    <p>И мне!</p>
    <p>И мне!</p>
</AttentionSeeker>
```

ГЛАВА 20



Прочие полезные инструменты и библиотеки

20.1. Библиотека React Window: вывод больших перечней

При выводе большого перечня каких-либо позиций для отображения каждой позиции создается отдельный элемент страницы (строка таблицы, пункт списка, абзац и др.). Все эти многочисленные элементы занимают большой объем оперативной памяти, а их вывод занимает много времени.

Повысить производительность приложения можно, выводя на страницу только часть позиций такого перечня, что потребует создания значительно меньшего количества элементов страницы. При прокрутке такого перечня для вывода последующих позиций будут использованы уже существующие элементы страницы. На странице создается своего рода «окно», через которое будет видна часть выводимого перечня[1].

Реализовать такой «оконный» вывод перечней — списков и таблиц — позволяет дополнительная библиотека React Window. Она устанавливается подачей команды:

```
npm install react-window
```

> ***На заметку***
>
> Полная документация по библиотеке React Window приведена по интернет-адресу: **https://react-window.vercel.app/**.

Все компоненты, предоставляемые этой библиотекой, объявлены в модуле `react-window`.

20.1.1. Компонент *FixedSizeList*: вывод списков с пунктами одинакового размера

Компонент `FixedSizeList` выводит перечни в виде списков, пункты которых имеют одинаковый размер: высоту — у вертикальных списков, ширину — у горизонтальных.

---

[1] Что отражено в названии описываемой далее библиотеки: window в переводе с английского — окно.

В качестве содержимого этого компонента указывается ссылка на другой React-компонент, формирующий отдельную позицию перечня (а не JSX-код, выводящий этот компонент!). Компонент-потомок должен принимать от компонента `FixedSizeList` следующие пропы:

- `index` — индекс выводимой позиции;
- `style` — стиль, формируемый компонентом `FixedSizeList`, указывающий местоположение текущей позиции и задаваемый у ее корневого тега;
- `isScrolling` — `true`, если в текущий момент производится прокрутка перечня, и `false` — если прокрутка не производится. Этот проп передается, только если проп `useIsScrolling` у компонента `FixedSizeList` имеет значение `true`. Может использоваться для вывода внутри позиции какого-либо индикатора прокрутки;
- `data` — произвольные данные, заданные в пропе `itemData` компонента `FixedSizeList`.

Помимо элементов, выводящих позиции, которые отображаются в «окне», компонент также формирует по нескольку (по умолчанию — по одному) элементов-позиций выше и ниже отображаемых. Эти дополнительные сгенерированные элементы нужны для предотвращения мерцания списка при его прокрутке.

Компонент `FixedSizeList` формируется двумя элементами (по умолчанию — блоками):

- внешним — создающим само «окно», в котором выводятся позиции перечня;
- внутренним — перемещающимся по вертикали или горизонтали (в зависимости от ориентации компонента) внутри внешнего элемента под элементами-позициями и обеспечивающим прокрутку перечня.

Компонент `FixedSizeList` поддерживает следующие пропы:

- `width` — ширина компонента. В зависимости от его ориентации указывается:
  - вертикальной — либо в виде целого числа в пикселах, либо в виде строки в процентах (например, `'90%'`);
  - горизонтальной — в виде целого числа в пикселах;
- `height` — высота компонента. В зависимости от его ориентации указывается:
  - вертикальной — в виде целого числа в пикселах;
  - горизонтальной — либо в виде целого числа в пикселах, либо в виде строки в процентах (например, `'50%'`);
- `itemCount` — общее количество позиций в выводимом перечне;
- `itemSize` — размер элемента, выводящего позицию (высота — у вертикального списка, ширина — у горизонтального). Задается в виде целого числа в пикселах.

Все эти пропы обязательны для указания. Следующие — необязательны:

- `layout` — ориентация выводимого списка в виде строкового обозначения: `'vertical'` (вертикальная) или `'horizontal'` (горизонтальная). По умолчанию: `'vertical'`;

- `itemData` — произвольные данные, которые будут переданы компонентам-позициям в пропе `data`. Как правило, с этим пропом передается сам выводимый перечень;
- `itemKey` — принимается во внимание, только если указан проп `itemData`. Задает функцию, возвращающую уникальный идентификатор очередной выводимой позиции. Эта функция должна принимать два параметра: индекс позиции и данные, присвоенные пропу `itemData`.

  По умолчанию используется функция, встроенная в саму библиотеку и возвращающая индекс позиции в перечне. Эта функция подходит только для следующих случаев: если выводимый перечень не изменяется, не сортируется и если компонент-потомок не содержит состояния. В противном случае следует указать свою функцию, выдающую какой-либо другой уникальный идентификатор;
- `useIsScrolling` — если `true`, компоненту-потомку будет дополнительно передаваться проп `isScrolling` (описан ранее), если `false` — не будет (по умолчанию: `false`);
- `overscanCount` — количество дополнительных элементов-позиций, генерируемых выше и ниже непосредственно выводящихся в «окне». Значение по умолчанию: 1. Если при прокрутке возникает мерцание, это количество следует увеличить;
- `initialScrollOffset` — изначальная позиция прокрутки в виде целого числа в пикселах (по умолчанию: `0`);
- `className` — дополнительные стилевые классы для внешнего элемента (по умолчанию: `undefined`, т. е. отсутствие дополнительных стилевых классов);
- `style` — дополнительный стиль для внешнего элемента (по умолчанию: `undefined`, т. е. отсутствие дополнительного стиля);
- `innerElementType` — имя тега, используемого для создания внутреннего элемента (по умолчанию: `'div'` — как правило, этот тег подходит в большинстве случаев);
- `outerElementType` — имя тега, используемого для создания внешнего элемента (по умолчанию: `'div'` — как правило, этот тег подходит в большинстве случаев);
- `onItemRendered` — функция, вызываемая при выводе в «окне» очередной порции элементов, отображающих позиции перечня. Должна принимать в качестве параметра простой объект со следующими свойствами:
  - `overscanStartIndex` — индекс первого сгенерированного элемента;
  - `overscanStopIndex` — индекс последнего сгенерированного элемента;
  - `visibleStartIndex` — индекс первого выведенного элемента;
  - `visibleStopIndex` — индекс последнего выведенного элемента.

  Значения всех свойств представляют собой целые числа. Результата функция возвращать не должна;

- `onScroll` — функция, вызываемая при изменении позиции прокрутки списка. Должна принимать в качестве параметра простой объект со следующими свойствами:
  - `scrollDirection` — направление прокрутки в виде строкового обозначения: `'forward'` (вперед) или `'backward'` (назад);
  - `scrollOffset` — дистанция прокрутки в виде целого числа в пикселах;
  - `scrollUpdateWasRequested` — `true`, если прокрутка была выполнена программно, `false` — если была произведена пользователем.

  Результата функция возвращать не должна;
- `innerRef` — реф, связываемый с внутренним элементом;
- `outerRef` — реф, связываемый с внешним элементом.

Вот пример кода, выводящего перечень дел:

```
import { FixedSizeList } from 'react-window';

// Компонент, выводящий отдельную позицию перечня
function TodoItem(props) {
    const item = props.data.list[props.index];

    return (
        <table style={props.style}>
            <tr>
                <td>
                    {item.title}
                </td>
                <td>
                    <button title="Пометить как выполненное"
                            onClick={() => props.data.setDone(item.key)}
                    >
                        &#9745;
                    </button>
                </td>
            </tr>
        </table>
    );
}

// Компонент, выводящий сам перечень дел
export default function TodoList(props) {
    return (
        <section>
            <FixedSizeList width="90%" height={200}
                           itemCount={props.list.length}
                           itemSize={100} itemData={props}
                           itemKey={(index, data) => data.list[index].key}
            >
```

```
                {TodoItem}
            </FixedSizeList>
        </section>
    );
}
```

Еще компонент `FixedSizeList` поддерживает два полезных метода:

- `scrollTo(<смещение>)` — прокручивает список на заданное *смещение*, которое должно задаваться в виде целого числа в пикселах. Смещение может быть как положительным, так и отрицательным;
- `scrollToItem(<индекс>[, <выравнивание>='auto'])` — прокручивает список так, чтобы позиция с заданным *индексом* стала видимой. *Индекс* позиции задается в виде целого числа.

  Также можно задать *выравнивание* позиции в «окне» в виде одного из следующих строковых обозначений:

  - `'auto'` — прокручивает список ровно настолько, чтобы позиция стала видимой. Если позиция уже видима, ничего не делает;
  - `'smart'` — если позиция полностью невидима, прокручивает список так, чтобы она располагалась в центре «окна». Если позиция видима частично, прокручивает список ровно настолько, чтобы она стала видимой. Если позиция уже видима, ничего не делает;
  - `'center'` — прокручивает список так, чтобы позиция располагалась в центре «окна»;
  - `'start'` — прокручивает список так, чтобы позиция располагалась в начале «окна» (если список вертикальный — вверху, если горизонтальный — слева);
  - `'end'` — прокручивает список так, чтобы позиция располагалась в конце «окна» (если список вертикальный — внизу, если горизонтальный — справа).

  Пример:

```
import { useRef } from 'react';
. . .
const listRef = useRef(undefined);
. . .
<FixedSizeList . . . ref={listRef}> . . . </FixedSizeList>
. . .
listRef.current.scrollToItem(2, 'smart');
```

20.1.2. Компонент *VariableSizeList*: вывод списков с пунктами разного размера

Компонент `VariableSizeList` аналогичен компоненту `FixedSizeList`, только выводит списки, пункты которых имеют разный размер.

Компонент `VariableSizeList` поддерживает те же самые пропы, что и `FixedSizeList`, и, кроме того, следующие:

- `itemSize` — функция, возвращающая размер (высоту — у вертикального списка, ширину — у горизонтального) очередного элемента-позиции. В качестве параметра функция должна принимать индекс очередной позиции. Проп обязателен к указанию.

  Размеры элементов, возвращенные функцией, сохраняются в оперативной памяти для повышения производительности;

- `estimatedItemSize` — приблизительный размер элемента, выводящего позицию. Задается в виде целого числа в пикселах. По умолчанию: 50.

Вот пример использования этого компонента (для вывода отдельной позиции применяется компонент `TodoItem`, код которого приведен в *разд. 20.1.1*):

```
import { VariableSizeList } from 'react-window';
. . .
<VariableSizeList width="90%" height={200}
                  itemCount={props.list.length}
                  estimatedItemSize={100} itemData={props}
                  itemKey={(index, data) => data.list[index].key}
                  itemSize={ind => props.list[ind].done ? 100 : 200}
>
    {TodoItem}
</VariableSizeList>
```

Компонент `VariableSizeList` поддерживает все методы, поддерживаемые `FixedSizeList`, и дополнительно метод `resetAfterIndex()`, который удаляет из памяти сохраненные ранее размеры элементов, начиная с указанного *индекса*:

```
resetAfterIndex(<индекс>[, <обновить компонент?>=true])
```

По умолчанию компонент обновляется после удаления сохраненных индексов. Чтобы отключить автоматическое обновление, следует передать с параметром *обновить компонент* значение `false`.

Метод `resetAfterIndex()` следует вызывать после изменения содержимого позиций выводимого перечня.

20.1.3. Компонент *FixedSizeGrid*: вывод таблиц с ячейками одинаковых размеров

Компонент `FixedSizeGrid` выводит большие перечни в виде таблиц, ячейки которых имеют одинаковые ширину и высоту.

В качестве содержимого этого компонента указывается ссылка на другой React-компонент, который будет выводить отдельную ячейку таблицы (а не JSX-код, выводящий этот компонент!). Компонент-потомок должен принимать от компонента `FixedSizeGrid` следующие пропы:

- `rowIndex` — индекс строки, в которой находится выводимая ячейка;
- `columnIndex` — индекс столбца, в котором находится выводимая ячейка;

- `style` — стиль, формируемый компонентом `FixedSizeGrid`, указывающий местоположение текущей ячейки и задаваемый у ее корневого тега;
- `isScrolling` — `true`, если в текущий момент производится прокрутка таблицы, и `false` — если прокрутка не производится. Этот проп передается, только если проп `useIsScrolling` у компонента `FixedSizeGrid` имеет значение `true`. Может использоваться для вывода внутри ячейки какого-либо индикатора прокрутки;
- `data` — произвольные данные, заданные в пропе `itemData` компонента `FixedSizeGrid`.

Компонент `FixedSizeGrid` генерирует по нескольку дополнительных элементов-строк выше и ниже выводимой, равно как и по нескольку элементов-столбцов левее и правее выводимых, — для предотвращения мерцания при прокрутке. Как и рассмотренные ранее компоненты-списки, он состоит из двух элементов: внешнего, формирующего «окно», и внутреннего, обеспечивающего прокрутку.

Компонент `FixedSizeList` поддерживает следующие пропы:

- `width` — ширина компонента в виде целого числа в пикселах;
- `height` — высота компонента в виде целого числа в пикселах;
- `rowCount` — общее количество строк в выводимой таблице;
- `columnCount` — общее количество столбцов в выводимой таблице;
- `rowHeight` — высота строк в виде целого числа в пикселах;
- `columnWidth` — ширина столбцов в виде целого числа в пикселах.

Все эти пропы обязательны для указания. Следующие — необязательны:

- `itemData` — произвольные данные, которые будут переданы компонентам-позициям в пропе `data`. Как правило, с этим пропом передается сам выводимый перечень;
- `itemKey` — принимается во внимание, только если указан проп `itemData`. Задает функцию, возвращающую уникальный идентификатор очередной выводимой ячейки. Эта функция должна принимать в качестве параметра простой объект со следующими свойствами:
  - `rowIndex` — индекс строки выводимой ячейки;
  - `columnIndex` — индекс столбца выводимой ячейки;
  - `data` — данные, присвоенные пропу `itemData`.

  По умолчанию используется функция, встроенная в саму библиотеку и возвращающая строковый идентификатор формата: *<индекс строки>:<индекс столбца>*. Эта функция подходит только для следующих случаев: если выводимый перечень не изменяется, не сортируется и если компонент-потомок не содержит состояния. В противном случае следует указать свою функцию, выдающую какой-либо другой уникальный идентификатор;
- `useIsScrolling` — если `true`, компоненту-потомку будет дополнительно передаваться проп `isScrolling` (описан ранее), если `false` — не будет (по умолчанию: `false`);

- `overscanRowCount` — количество дополнительных элементов-строк, генерируемых выше и ниже непосредственно выводящихся в «окне». Значение по умолчанию: 1. Если при прокрутке возникает мерцание, это количество следует увеличить;
- `overscanColumnCount` — количество дополнительных элементов-столбцов, генерируемых левее и правее выводящихся в «окне». Значение по умолчанию: 1. Если при прокрутке возникает мерцание, это количество следует увеличить;
- `initialScrollTop` — изначальная позиция прокрутки по вертикали в виде целого числа в пикселах (по умолчанию: `0`);
- `initialScrollLeft` — изначальная позиция прокрутки по горизонтали в виде целого числа в пикселах (по умолчанию: `0`);
- `className` — дополнительные стилевые классы для внешнего элемента (по умолчанию: `undefined`, т. е. отсутствие дополнительных стилевых классов);
- `style` — дополнительный стиль для внешнего элемента (по умолчанию: `undefined`, т. е. отсутствие дополнительного стиля);
- `innerElementType` — имя тега, используемого для создания внутреннего элемента (по умолчанию: `'div'` — как правило, этот тег подходит в большинстве случаев);
- `outerElementType` — имя тега, используемого для создания внешнего элемента (по умолчанию: `'div'` — как правило, этот тег подходит в большинстве случаев);
- `onItemRendered` — функция, вызываемая при выводе в «окне» очередной порции элементов, отображающих ячейки таблицы. Должна принимать в качестве параметра простой объект со следующими свойствами:
  - `overscanColumnStartIndex` — индекс первого сгенерированного столбца;
  - `overscanColumnStopIndex` — индекс последнего сгенерированного столбца;
  - `overscanRowStartIndex` — индекс первой сгенерированной строки;
  - `overscanRowStopIndex` — индекс последней сгенерированной строки;
  - `visibleColumnStartIndex` — индекс первого выведенного столбца;
  - `visibleColumnStopIndex` — индекс последнего выведенного столбца;
  - `visibleRowStartIndex` — индекс первой выведенной строки;
  - `visibleRowStopIndex` — индекс последней выведенной строки.

  Значения всех свойств представляют собой целые числа. Результата функция возвращать не должна;
- `onScroll` — функция, вызываемая при изменении позиции прокрутки таблицы. Должна принимать в качестве параметра простой объект со следующими свойствами:
  - `horizontalScrollDirection` — направление прокрутки по горизонтали в виде строкового обозначения: `'forward'` (вперед) или `'backward'` (назад);

- `verticalScrollDirection` — направление прокрутки по вертикали в виде строкового обозначения: `'forward'` (вперед) или `'backward'` (назад);
- `scrollLeft` — дистанция прокрутки по горизонтали в виде целого числа в пикселах;
- `scrollTop` — дистанция прокрутки по вертикали в виде целого числа в пикселах;
- `scrollUpdateWasRequested` — `true`, если прокрутка была выполнена программно, `false` — если была произведена пользователем.

Результата функция возвращать не должна;

- `innerRef` — реф, связываемый с внутренним элементом;
- `outerRef` — реф, связываемый с внешним элементом.

Листинг 20.1 содержит код, выводящий таблицу с результатами деления чисел от 1 до 25 друг на друга.

Листинг 20.1. Использование компонента `FixedSizeGrip`

```
import { FixedSizeGrid } from 'react-window';

function Cell(props) {
    let output;
    if (props.rowIndex === 0)
        if (props.columnIndex === 0)
            output = '';
        else
            output = props.columnIndex;
    else
        if (props.columnIndex === 0)
            output = props.rowIndex;
        else
            output = props.columnIndex / props.rowIndex;

    return <div style={props.style}>{output}</div>;
}

export default function App() {
    return (
        <FixedSizeGrid rowCount={26} columnCount={26}
                       rowHeight={40} columnWidth={200}
                       width={400} height={400}
        >
            {Cell}
        </FixedSizeGrid>
    );
}
```

Еще компонент `FixedSizeGrid` поддерживает два полезных метода:

- `scrollTo(<смещения>)` — прокручивает таблицу на заданные *смещения*. Последние указываются в виде простого объекта со свойствами `scrollLeft` (смещение по горизонтали) и `scrollTop` (смещение по вертикали). Значения этих свойств задаются в виде целых чисел в пикселах. Пример:

```
import { useRef } from 'react';
. . .
const gridRef = useRef(undefined);
. . .
<FixedSizeGrid . . . ref={gridRef}> . . . </FixedSizeGrid>
. . .
gridRef.current.scrollTo({ scrollLeft: 300, scrollTop: 80 });
```

- `scrollToItem(<местоположение>)` — прокручивает таблицу так, чтобы заданное *местоположение* появилось в области видимости. *Местоположение* задается в виде простого объекта со свойствами: `rowIndex` (индекс строки), `columnIndex` (индекс столбца) и `align` (выравнивание, аналогично второму параметру одноименного метода списков, который описан в *разд. 20.1.1*):
  - если заданы индексы строки и столбца — таблица прокручивается таким образом, чтобы стала видимой ячейка, находящаяся на пересечении строки и столбца с указанными индексами;
  - если задан только индекс строки — таблица прокручивается до этой строки;
  - если задан только индекс столбца — таблица прокручивается до этого столбца;
  - если не задан ни один из индексов — ничего не происходит.

20.1.4. Компонент *VariableSizeGrid*: вывод таблиц с ячейками разных размеров

Компонент `VariableSizeGrid` аналогичен компоненту `FixedSizeGrid`, только выводит таблицы, строки которых имеют разную высоту, а столбцы — разную ширину.

Компонент `VariableSizeGrid` поддерживает те же самые пропы, что и `FixedSizeGrid`, и, кроме того, следующие:

- `rowHeight` — функция, возвращающая высоту очередной строки таблицы. В качестве параметра она должна принимать индекс строки;
- `columnWidth` — функция, возвращающая ширину очередного столбца таблицы. В качестве параметра она должна принимать индекс столбца.

  Размеры строк и столбцов, возвращенные функциями, сохраняются в оперативной памяти для повышения производительности.

  Оба эти пропа обязательны к указанию. Следующие пропы необязательны:

- `estimatedRowHeight` — приблизительная высота строки. Задается в виде целого числа в пикселах. По умолчанию: 50;

- `estimatedColumnWidth` — приблизительная ширина столбца. Задается в виде целого числа в пикселах. По умолчанию: 50.

Вот пример использования этого компонента (для вывода отдельной позиции применяется компонент `Cell` из листинга 20.1):

```
import { VariableSizeGrid } from 'react-window';
. . .
<VariableSizeGrid rowCount={26} columnCount={26}
                  estimatedRowHeight={40}
                  estimatedColumnWidth={200}
                  rowHeight={index => 40}
                  columnWidth={index => index == 0 ? 50 : 200}
                  width={400} height={400}
>
    {Cell}
</VariableSizeGrid>
```

Компонент `VariableSizeGrid` поддерживает все методы компонента `FixedSizeGrid`, а также следующие методы:

- `resetAfterRowIndex()` — удаляет из памяти сохраненные ранее значения высоты строк, начиная со строки с заданным *индексом*:

  `resetAfterRowIndex(<`*индекс*`>[, <`*обновить компонент?*`>=true])`

  По умолчанию компонент обновляется после удаления сохраненных индексов. Чтобы отключить автоматическое обновление, следует передать с параметром *обновить компонент* значение `false`;

- `resetAfterColumnIndex()` — удаляет из памяти сохраненные ранее значения ширины столбцов, начиная со столбца с заданным индексом. Формат вызова такой же, как и у метода `resetAfterRowIndex()`;

- `resetAfterIndices(<`*настройки*`>)` — удаляет из памяти сохраненные ранее значения высоты строк и ширины столбцов. В качестве *настроек* указывается простой объект со следующими свойствами:

  - `rowIndex` — индекс строки, начиная с которой следует удалить все сохраненные значения высоты;
  - `columnIndex` — индекс столбца, начиная с которого следует удалить все сохраненные значения ширины;
  - `shouldForceUpdate` — если `true`, после удаления сохраненных значений компонент будет обновлен, если `false` — не будет (по умолчанию: `true`).

Эти методы следует вызывать после изменения содержимого позиций, присутствующих в выводимой таблице.

20.2. Работа с метатегами

В React 19 появилась возможность указывать в составе любого компонента метатеги, задающие название веб-страницы, метаданные, привязывающие к странице таблицы стилей и сторонние веб-сценарии. Такой метатег можно поместить в произвольном месте компонента — React самостоятельно перенесет его в секцию заголовка страницы (тег `<head>`).

20.2.1. Указание названия веб-страницы (тег *<title>*)

Для указания названия веб-страницы следует использовать привычный парный тег `<title>`. Никаких специфических пропов он не поддерживает. Текст названия помещается внутри этого тега.

Как говорилось ранее, тег `<title>` (как и прочие метатеги) можно поместить в произвольном месте компонента, а React самостоятельно перенесет этот тег в секцию заголовка страницы. Тег `<title>`, записанный в коде страницы приложения public\index.html (см. *разд. 3.3.1*), при этом сохранится.

Пример:

```
export default function TodoList() {
    . . .
    return (
        <section>
            <title>Перечень дел :: Todos</title>
            <h1>Дела</h1>
            . . .
        </section>
    );
}
```

Текст заголовка должен представлять собой одну строку:

```
export default function TodoDetail() {
    . . .
    return (
        <section>
            <title>{`${todo.title} :: Перечень дел :: Todos`}</title>
            . . .
        </section>
    );
}
```

Указание его в виде комбинации обычных строк и языковых конструкций `{<выражение>}`, например:

```
<title>{todo.title} :: Перечень дел :: Todos</title>
```

приведет к некорректному выводу текста заголовка.

Внимание!

Во всех компонентах, в текущий момент выведенных на странице, должен присутствовать лишь один тег `<title>`. Если в разных компонентах имеются два или более таких тега, все они будут помещены в секцию заголовка страницы. Какой из тегов будет использован веб-обозревателем, предсказать невозможно.

Кроме того, тег `<title>`, присутствующий в странице приложения, следует удалить — чтобы он не «вводил в заблуждение» веб-обозреватель.

20.2.2. Указание метаданных (тег *<meta>*)

Для задания метаданных страницы, как и при обычной веб-верстке, используется тег `<meta>`. Никаких специфических пропов он не поддерживает.

Все теги `<meta>`, присутствующие во всех компонентах, которые выведены на странице, будут перенесены в секцию заголовка страницы. Теги, записанные в коде страницы приложения public\index.html (см. *разд. 3.3.1*), при этом сохранятся.

Пример:

```
export default function App() {
    . . .
    return (
        <div className="container">
            <meta name="author" content="Читатель книги" />
            <meta name="keywords"
                  content="React, JavaScript, веб-приложение" />
            <meta name="description"
                  content="Учебное веб-приложение: список планируемых дел" />
            . . .
        </div>
    );
}
```

20.2.3. Привязка к веб-странице сторонних файлов (тег *<link>*)

Для привязки к странице сторонних файлов служит тег `<link>`. Разновидность привязываемого файла (таблица стилей, значок сайта и др.) указывается в его свойстве `rel`. Кроме того, тегом поддерживаются свойства `href`, `crossOrigin`, `referrerPolicy`, `fetchPriority`, `hrefLang`, `integrity` и `type`, соответствующие одноименным атрибутам.

Все теги `<link>`, присутствующие во всех компонентах, которые выведены на странице, будут перенесены в секцию заголовка страницы с сохранением порядка их следования (исключения из этого правила описаны далее). Дубликаты тегов, привязывающие один и тот же файл, — удалены. Теги, в которых указаны таблицы стилей, — переупорядочены согласно значениям пропа `precedence` (см. далее). Теги `<link>`, записанные в коде страницы приложения, сохранятся.

В случае указания в пропе `rel` тега `<link>` значения:

- `"stylesheet"` (привязка таблицы стилей) — также поддерживаются свойства `media` и `title`, соответствующие одноименным атрибутам тега `<link>`.

  Дополнительно поддерживаются следующие пропы:

  - `precedence` — обозначение приоритета текущей таблицы стилей в виде строки: `"reset"` (самый низкий), `"low"` (низкий), `"medium"` (средний) или `"high"` (высокий).

    Теги `<link>` с таблицами стилей при перемещении в секцию заголовка страницы будут переупорядочены согласно значениям этого пропа. Теги с более высоким приоритетом — помещены после низкоприоритетных тегов.

    Если проп `precedence` не задан, тег `<link>` не будет перемещен в секцию заголовка страницы и останется на месте, в составе компонента[1].

    Пример:

```
export default function App() {
    . . .
    return (
        <div className="container">
            <link rel="stylesheet" href="navbar.css"
                                   precedence="high" />
            <link rel="stylesheet" href="layout.css"
                                   precedence="medium" />
            <link rel="stylesheet" href="base.css"
                                   precedence="reset" />
            . . .
        </div>
    );
}
// Результат:
// <head>
//     . . .
//     <link rel="stylesheet" href="base.css">
//     <link rel="stylesheet" href="layout.css">
//     <link rel="stylesheet" href="navbar.css">
// </head>
```

  - `disabled` — если `false`, таблица стилей будет загружена и привязана к странице, если `true` — не будет (по умолчанию: `false`);
  - `onLoad` — функция, которая будет вызвана после успешной загрузки таблицы стилей. Задаваемая функция не должна ни принимать параметров, ни возвращать результата;

[1] Тем не менее указанная в нем таблица стилей все равно будет загружена.

- `onError` — функция, которая будет вызвана, если при загрузке таблицы стилей возникнет ошибка. Задаваемая функция не должна ни принимать параметров, ни возвращать результата.

  В случае указания любого из последних трех пропов, тег `<link>` не будет перемещен в секцию заголовка страницы, а останется на месте;

- `"icon"` или `"apple-touch-icon"` (задание значка сайта) — дополнительно доступно для указания свойство `sizes`, соответствующее одноименному атрибуту тега `<link>`:

```
<link rel="icon" href="favicon.ico" />
```

- `"preload"` или `"modulepreload"` (предварительная загрузка файлов) — также можно указать свойства `as`, `imageSrcSet` и `imageSizes`, соответствующие одноименным атрибутам тега `<link>`:

```
<link rel="preload" href="background1.png"
                    imageSizes="(max-width: 600px) 480px, 800px" />
<link rel="preload" imageSrcSet="background2-320w.png, ↳
                                 background2-480w.png 1.5x, ↳
                                 background2-640w.png 2.0x " />
```

20.2.4. Задание внутренних таблиц стилей

Внутренняя таблица стилей создается тегом `<style>`. Он поддерживает свойства `media`, `nonce` и `title`, соответствующие одноименным атрибутам тега. Внутри тега записывается CSS-код таблицы стилей.

Все теги `<style>`, присутствующие во всех компонентах и содержащие проп `precedence`, будут перенесены в секцию заголовка страницы и переупорядочены согласно значениям упомянутого пропа. Теги, содержащие дублирующиеся значения в пропе `href`, — удалены. Теги `<style>`, записанные в коде страницы приложения, сохранятся.

Тег `<style>` поддерживает следующие специфические пропы:

- `precedence` — обозначение приоритета текущей таблицы (описан в *разд. 20.2.3*);
- `href` — уникальный идентификатор таблицы стилей в виде произвольной строки. Если имеется несколько тегов `<style>` с одинаковым идентификатором, в секцию заголовка будет помещен лишь один экземпляр такого тега.

Пример:

```
<style href="internal-style-sheet" precedence="medium">
    h1.heading {
        font-size: 48pt;
        font-weight: normal;
    }
</style>
```

20.2.5. Задание сторонних веб-сценариев

При необходимости использовать React совместно с какой-либо другой библиотекой, написанной на JavaScript, файл с этой библиотекой можно привязать к странице посредством тега `<script>`. Он поддерживает свойства `src`, `async`, `crossOrigin`, `fetchPriority`, `integrity`, `noModule`, `nonce`, `referrer` и `type`, соответствующие одноименным атрибутам тега.

В теге `<script>` можно задать веб-сценарий:

- внешний — записав его интернет-адрес в свойстве `src`:

```
<script async src="very-cool-library.js" />
```

- внутренний — записав его код непосредственно в теге:

```
<script async>
    . . .
</script>
```

Все теги `<script>`, присутствующие во всех компонентах, содержащие проп `async` со значением `true` и *не* содержащие пропов `onLoad` и `onError`, будут перенесены в секцию заголовка страницы. Теги, содержащие дублирующиеся веб-сценарии, удалены. Теги `<script>`, записанные в коде страницы приложения, сохранятся.

Тег `<script>` поддерживает дополнительные пропы `onLoad` и `onError`, описанные в *разд. 20.2.3*.

20.3. Предварительная загрузка интернет-ресурсов

React 19 также предлагает инструменты для предварительной загрузки интернет-ресурсов, включая файлы со сторонними таблицами стилей и веб-сценариями. Применение предварительной загрузки позволит ускорить запуск приложения.

Для предварительной загрузки применяются следующие функции, объявленные в модуле `react-dom`:

- `preload()` — предварительная загрузка интернет-ресурса (графического изображения, аудио, видеоролика, шрифта и др.), находящегося по заданному *интернет-адресу*:

  `preload(<интернет-адрес>, <настройки>)`

  Интернет-адрес указывается в виде строки. *Настройки* должны быть заданы в виде простого объекта со свойствами:

  - `as` — аналогичен атрибуту `as` тега `<link>`. Единственная настройка, обязательная к указанию;
  - `crossOrigin`, `referrerPolicy`, `integrity`, `type`, `nonce`, `fetchPriority`, `imageSrcSet` и `imageSizes` — аналогичны одноименным атрибутам тега `<link>`.

Пример предварительной загрузки изображения:

```
import { preload } from 'react-dom';
export default function Logo() {
    preload('http://www.someserver.ru/images/logo.png', { as: 'image' });
    . . .
    return (
        . . .
        <div><img src="http://www.someserver.ru/images/logo.png" /></div>
        . . .
    );
}
```

Пример предварительной загрузки внешней таблицы стилей перед переходом на экран, который ее использует:

```
// Модуль src\App.js
import { preload } from 'react-dom';
export default function App() {
    . . .
    const handleTodoAddClick = () => {
        preload('http://www.otherserver.ru/styles/todo-add.css',
                { as: 'style' });
        navigate('/add');
    };
    . . .
    return (
        . . .
        <input type="button" value="Добавить дело"
                             onClick={handleTodoAddClick} />
        . . .
    );
}

// Модуль src\TodoAdd.js
export default function TodoAdd() {
    . . .
    return (
        . . .
        <link rel="stylesheet"
              href="http://www.otherserver.ru/styles/todo-add.css"
              precedence="medium" />
        . . .
    );
}
```

- `preloadModule()` — максимально быстрая предварительная загрузка программного модуля стандарта ECMAScript, расположенного по заданному *интернет-адресу*:

```
preloadModule(<интернет-адрес>, <настройки>)
```

Интернет-адрес указывается в виде строки. *Настройки* должны быть заданы в виде простого объекта со свойствами:

- `as` — должно содержать значение `'script'`;
- `crossOrigin`, `integrity` и `nonce` — аналогичны одноименным атрибутам тега `<link>`.

Пример:

```
// Модуль src\App.js
import { preloadModule } from 'react-dom';
export default function App() {
    . . .
    const handleStatClick = () => {
        preloadModule('http://www.otherserver.ru/scripts/chart.js',
                      { as: 'script' });
        navigate('/stat');
    };
    . . .
    return (
        . . . .
        <input type="button" value="Статистика"
                             onClick={handleStatClick} />
        . . .
    );
}

// Модуль src\Stat.js
export default function Stat() {
    . . .
    return (
        . . .
        <script async src="http://www.otherserver.ru/scripts/chart.js" />
        . . .
    );
}
```

- `preinit()` — то же самое, что и `preload()`, только предварительно загруженную таблицу стилей сразу же применяет к странице, а предварительно загруженный веб-сценарий — выполняет. В свойстве `as` объекта *настроек* можно указать лишь значения `'style'` и `'script'`. Пример:

```
import { preinit } from 'react-dom';
export default function TodoAdd() {
    preinit('http://www.otherserver.ru/styles/todo-add.css',
            { as: 'style' });
    . . .
}
```

- `preinitModule()` — то же самое, что и `preloadModule()`, только предварительно загруженный модуль сразу же выполняет:

```
import { preinitModule } from 'react-dom';
export default function Stat() {
    preinitModule('http://www.otherserver.ru/scripts/chart.js',
                  { as: 'script' });
    . . .
}
```

- `preconnect(<интернет-адрес>)` — установка соединения с сервером, находящимся по указанному *интернет-адресу*. *Интернет-адрес* задается в виде строки. Пример:

```
import { preconnect } from 'react-dom';
export default function ShowResourcesFromOtherServer() {
    preconnect('http://www.someserver.ru');
    . . .
    return (
        . . .
        <link rel="stylesheet"
              href="http://www.someserver.ru/styles/layout.css"
              precedence="medium" />
        <script async src="http://www.someserver.ru/scripts/effects.js" />
        <div><img src="http://www.someserver.ru/images/logo.png" /></div>
        . . .
    );
}
```

Предварительная установка соединения с сервером уместна, если требуется загрузить несколько ресурсов, и позволяет ускорить их загрузку. Однако следует иметь в виду, что каждое установленное таким образом соединение отнимает системные ресурсы;

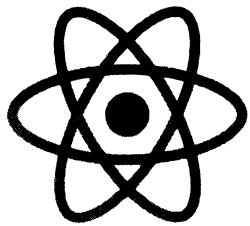
- `prefetchDNS(<доменное имя>)` — получение IP-адреса сервера, имеющего указанное *доменное имя*. *Доменное имя* задается в виде строки. Пример:

```
import { prefetchDNS } from 'react-dom';
. . .
export default function ShowResourcesFromManyServers() {
    prefetchDNS('http://www.someserver1.ru');
    prefetchDNS('http://www.someserver2.ru');
    prefetchDNS('http://www.someserver3.ru');
    . . .
}
```

Предварительное получение IP-адреса может принести пользу, если имеется необходимость загрузить множество ресурсов с нескольких серверов, и позволяет, ускорив их загрузку, также сэкономить системные ресурсы (поскольку соединения с этими серверами не устанавливаются).

Повторные вызовы любой из приведенных функций с указанием того же *интернет-адреса* будут проигнорированы.

ГЛАВА 21



Обработка ошибок

21.1. Обработка ошибок средствами JavaScript

Для реагирования на возникновение ошибок следует использовать обработчики исключений. В случае возникновения ошибки необходимо вывести на страницу какое-либо оповещение или даже «запасной» пользовательский интерфейс (скажем, кнопку для выполнения повторной попытки подключения к бэкенду). Пример:

```
export default function SomeComponent() {
    const [isError, setIsError] = useState(false);
    . . .
    const someFunction = () => {
        . . .
        try {
            // Выполняем операцию, которая может привести к возникновению
            // ошибки
        }
        catch(exc) {
            setIsError(true);
        }
    }
    . . .
    if (isError)
        return <p>Возникла ошибка!</p>;
    else
        return (
            // Выводим основной интерфейс
        );
}
```

21.2. Обработка ошибок средствами React Router

Новые версии библиотеки React Router предоставляют исключительно удобный инструментарий для обработки ошибок.

Все описанные далее хуки и функции объявлены в модуле `react-router-dom`.

21.2.1. Базовые средства обработки ошибок

Компонент `Route`, представляющий маршрут (см. *разд. 11.2.1*), поддерживает проп `errorElement`. В нем указывается JSX-код, выводящий компонент, в котором будут отображаться сообщения об ошибках, возникающих в целевых компонентах, загрузчиках и воздействиях, записанных в текущем маршруте и во всех вложенных в него маршрутах.

Вот пример задания компонента `ErrorMessage`, в котором будут выводиться сообщения об ошибках, во «внешнем» маршруте:

```
const rm = (
    <Route path="/" element={<App />} errorElement={<ErrorMessage />}>
        <Route index element={<TodoList />} />
        . . .
    </Route>
);
```

Получить в компоненте, который выводит сообщения об ошибках, объект исключения, представляющий возникшую ошибку, можно, вызвав хук `useRouteError()` без параметров.

В листинге 21.1 показан код компонента `ErrorMessage`, выводящего сообщения об ошибках. Компонент является дружественным к посетителю: получив исключение с текстовым сообщением `'notfound'`, он выводит развернутое оповещение об отсутствующем деле.

Листинг 21.1. Код компонента `ErrorMessage`

```
import { Link, useRouteError } from 'react-router-dom';

export default function ErrorMessage() {
    const error = useRouteError();

    return (
        <section>
            {error.message == 'notfound' &&
                <>
                    <h1>Дело не существует</h1>
                    <p>
```

```
                        Вернитесь на <Link to="/">перечень дел</Link>
                        и выберите какое-либо другое.
                    </p>
                </>
            }
            {error.message != 'notfound' &&
                <>
                    <h1>Ошибка</h1>
                    <p>{error.message}</p>
                </>
            }
        </section>
    );
}
```

Сообщить посетителю о том, что выбранное им дело не существует, можно, возбудив в коде загрузчика исключение с текстовым сообщением `'notfound'`:

```
export async function getTodo(key) {
    const todo = await loadTodo(key);
    if (!todo)
        throw new Error('notfound');
    return todo;
}
```

Компонент `Route` также поддерживает проп `ErrorBoundary` (с большой буквы!), в котором указывается ссылка на функцию, реализующую компонент, в котором будут выводиться сообщения о ошибках:

```
<Route path="/" element={<App />} ErrorBoundary={ErrorMessage}>
    . . .
</Route>
```

> ***Внимание!***
>
> Если в маршруте присутствуют пропы `errorElement` и `ErrorBoundary`, будет принят во внимание только проп `ErrorBoundary`.

Компонент, выводящий сообщения об ошибках, следует указать, по крайней мере, в наиболее «внешнем» из узловых маршрутов. Будучи записанным там, компонент станет выводить сообщения обо всех ошибках, возникающих в приложении.

В списке маршрутов можно записать несколько компонентов, выводящих сообщения об ошибках, — поставив их в разных маршрутах:

```
const rm = (
    <Route path="/" element={<App />} errorElement={<ErrorMessage />}>
        <Route index element={<TodoList />} loader={getTodos} />
        <Route path=":key" element={<TodoDetail />}
               loader={getTodo} action={actTodo}
```

```
                errorElement={<Error404 />} />
        . . .
    </Route>
);
```

В результате компонент `Error404`, записанный в маршруте с шаблонным путем **/:key**, станет выводить сообщения об ошибках, возникающих в компоненте `TodoDetail`, загрузчике `getTodo()` и воздействии `actTodo()`, записанных в том же маршруте. В то же время компонент `ErrorMessage`, указанный в узловом маршруте, выведет сообщения об ошибках, возникающих во всех остальных маршрутах.

21.2.2. Обработка HTTP-ошибок

HTTP-ошибка — это ошибка, представленная не обычным исключением JavaScript, а серверным ответом — объектом класса `Response`. Представление ошибок в таком виде позволяет передать компоненту, выводящему сообщения о них, дополнительные сведения о возникшей ошибке (например, код статуса ответа).

Вот пример оповещения посетителя о несуществующем деле путем возбуждения HTTP-ошибки вместо исключения JavaScript:

```
export async function getTodo(key) {
    const todo = await loadTodo(key);
    if (!todo)
        throw new Response('', { status: 404 });
    return todo;
}
```

Получить объект серверного ответа, представляющий HTTP-ошибку, можно вызовом уже знакомого нам хука `useRouteError()`.

Может возникнуть необходимость выяснить, является ли заданная *ошибка* обычным исключением или же HTTP-ошибкой. Помощь в этом окажет функция `isRouteErrorResponse(<ошибка>)`. Она вернет `true`, если ей была передана HTTP-ошибка, и `false` — в случае передачи обычного исключения.

Листинг 21.2 содержит код компонента `ErrorMessage2`, аналогичного `ErrorMessage` (см. листинг 21.1), только обрабатывающего HTTP-ошибки.

Листинг 21.2. Код компонента `ErrorMessage2`

```
import { Link, useRouteError, isRouteErrorResponse }
       from 'react-router-dom';

export default function ErrorMessage2() {
    const error = useRouteError();

    return (
        <section>
```

```
            {(isRouteErrorResponse(error) && error.status == 404) &&
                <>
                    <h1>Дело не существует</h1>
                    <p>
                        Вернитесь на <Link to="/">перечень дел</Link> и
                        выберите какое-либо другое.
                    </p>
                </>
            }
            {!isRouteErrorResponse(error) &&
                <>
                    <h1>Ошибка</h1>
                    <p>{error.message}</p>
                </>
            }
        </section>
    );
}
```

21.2.3. Обработка ошибок в приостановщиках *Await*

Приостановщик `Await` также поддерживает проп `errorElement`, описанный в *разд. 21.2.1*. Указанный в нем компонент будет выводить сообщения об ошибках, которые возникают в асинхронных загрузчике и воздействии, заданных в соответствующем маршруте, и приводят к отклонению возвращаемых ими промисов.

Пример:

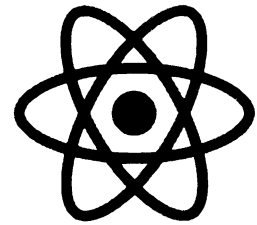
```
<Await resolve={todoPromise} errorElement={<AsyncErrorMessage />}>
    {todo => (
        . . .
    )}
</Await>
```

В компоненте, выводящем сообщение об ошибке, получить объект ошибки можно вызовом хука `useAsyncError()` без параметров.

Пример:

```
import { useAsyncError } from 'react-router-dom';
export default function AsyncErrorMessage() {
    const error = useAsyncError();
    return <p>{error.message}</p>
}
```

ГЛАВА 22



Разделение кода и загрузка по запросу

При запуске проекта на выполнение сборщик Webpack, входящий в состав React-проекта, собирает весь программный код в связку. Такая связка состоит из нескольких файлов-чанков, в одном из которых содержится весь код, написанный разработчиком веб-приложения, и библиотечный код (подробности — в *разд. 3.4*).

Код сложного приложения может оказаться очень большим, и содержащий его чанк будет загружаться долго. Во время загрузки посетителю придется наблюдать пустую страницу приложения, ожидая, когда на ней хоть что-то появится...

Ускорить запуск приложения можно, указав сборщику вынести некоторые JavaScript-модули в отдельные чанки. Обычно так поступают с редко используемым кодом (например, кодом экранов сведений о приложении, его разработчиках и т. п.). В результате размер главного чанка уменьшится, и он станет загружаться быстрее.

Чтобы использовать сущность из модуля, хранящегося в отдельном чанке, достаточно произвести операцию динамического импорта этой сущности. В результате соответствующий чанк будет загружен автоматически.

Более того, сама операция динамического импорта, присутствующая в коде, служит указанием сборщику Webpack вынести модуль, из которого выполняется динамический импорт, в отдельный чанк — благо сборщик уже настроен нужным образом.

Процесс выноса модулей в отдельные чанки носит название *разделения кода*, а импорт компонента из отдельного чанка — *загрузки по запросу*.

22.1. Загрузка по запросу средствами JavaScript

Базовые инструменты для загрузки по запросу предоставляет сам язык JavaScript. Это стандартная функция динамического импорта `import()`:

```
import(<путь к импортируемому модулю>)
```

Функция возвращает промис, подтверждаемый при успешной загрузке модуля с указанным *путем*. В качестве нагрузки промис содержит простой объект со всеми сущностями, экспортированными из загруженного модуля. Если импорт не увенчался успехом, промис будет отклонен и получит в качестве нагрузки объект исключения.

Пример:

```
import('./someModule.js')
    .then(imported => {imported.someFunction()})
    .catch(err => {console.log(err.message)});
```

Функция `import()` подходит для разделения кода с объявлениями обычных функций и классов. Для разделения кода, объявляющего компоненты, следует воспользоваться инструментами, описываемыми далее.

22.2. Загрузка по запросу средствами React

React предоставляет собственные средства загрузки по запросу, являющиеся надстройкой над функцией `import()` и работающие с модулями, которые содержат код компонентов.

Для загрузки по запросу таких модулей следует использовать функцию `lazy()` из модуля `react`:

```
lazy(<импортирующая функция>)
```

Заданная *импортирующая функция* должна вызывать функцию `import()`, импортирующую модуль с нужным компонентом, и возвращать результат вызова этой функции. Принимать параметры она не должна.

Функция `lazy()` возвращает объект модуля, который содержит код загруженного компонента.

Для вывода компонента, загруженного по запросу посредством функции `lazy()`, необходимо использовать уже знакомый нам компонент `Suspense` из модуля `react` (см. *разд. 9.2.1*). Выводимый компонент указывается в качестве его потомка. В пропе `fallback` можно задать какое-либо замещающее содержимое, которое будет отображаться на странице, пока импортируемый компонент загружается.

Вот пример загрузки по запросу компонента `TodoDetail`:

```
import { lazy, Suspense } from 'react';
const TodoDetail = lazy(() => import('./TodoDetail.js'));
. . .
<Route path="/" element={<App />}>
    . . .
    <Route path=":key" element={(
              <Suspense fallback={<p>Идет загрузка...</p>}>
                  <TodoDetail />
```

```
                </Suspense>
            )}
            . . . />
    . . .
</Route>
```

22.3. Загрузка по запросу средствами React Router

Новые версии библиотеки React Router позволяют вынести код компонента, загрузчика и воздействия, указанных в маршруте, в отдельный модуль. Этот модуль будет загружен по запросу при совпадении маршрута.

Модуль, загружаемый по запросу, должен содержать следующие функции, имеющие строго определенные имена и подвергнутые множественному экспорту;

- `element()` — возвращает содержимое, выводимое при совпадении текущего маршрута;
- `Component()` (с большой буквы!) — реализует компонент, выводимый при совпадении текущего маршрута.

  Если в модуле имеются обе функции: `element()` и `Component()` — будет принята к исполнению функция `Component()`.
- `loader()` — реализует загрузчик;
- `action()` — реализует воздействие;
- `errorElement()` — возвращает содержимое, выводимое при возникновении ошибки;
- `ErrorBoundary()` — реализует компонент, выводящий сообщение о возникшей ошибке.

  Если в модуле указаны обе функции: `errorElement()` и `ErrorBoundary()` — будет вызвана `ErrorBoundary()`.

Как можно видеть, все эти функции аналогичны пропам компонента `Route` (см. *разд. 11.2.1*, *12.1.1.2*, *12.1.2.2* и *21.2.1*). Каждая из этих функций, как и соответствующий ей проп, является необязательной для объявления (так, если целевой компонент не загружает никаких данных, то писать для него загрузчик не нужно).

Вот пример модуля, содержащего целевой компонент выбранного дела, загрузчик, загружающий выбранное дело с бэкенда, воздействие, помечающее дело как выполненное или удаляющее его (в зависимости от использованного HTTP-метода), и компонент для вывода сообщений об ошибках:

```
export function Component() {
    const todo = useLoaderData();
    return (
        <section>
            <h1>{todo.title}</h1>
```

```
            <p>{todo.createdAt}</p>
            . . .
        </section>
    );
}

export async function loader({ params }) {
    . . .
}

export function action({ params, request }) {
    . . .
}

export function ErrorBoundary() {
    return (
        <section>
            <h1>Дело не существует</h1>
        </section>
    );
}
```

Для указания такого модуля в маршруте применяется проп `lazy` компонента `Route`. В нем записывается импортирующая функция, аналогичная задаваемой в функции `lazy()` (см. *разд. 22.2*). Пример:

```
<Route path=":key" lazy={() => import('./TodoDetail.js')} />
```

Совершенно необязательно записывать всю необходимую функциональность в модуле, загружаемом по запросу. Часть функциональности можно вынести в маршрут.

Вот пример записи обоих компонентов в модуле, загружаемом по запросу, а загрузчика и воздействия — в маршруте:

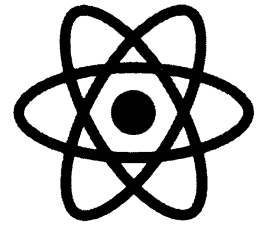
```
// Модуль src\TodoDetail.js
export function Component() {
    . . .
}

export function ErrorBoundary() {
    . . .
}

// Модуль src\routes.js
. . .
import { getTodo, actTodo } from "./api.js";
. . .
<Route path=":key" lazy={() => import('./TodoDetail.js')}
       loader={getTodo} action={actTodo} />
```

Какой подход применить на практике — дело вкуса.

ГЛАВА 23



Расширенные средства отладки

Базовые средства выявления ошибок, описанные в *разд. 3.5*, безусловно, помогут на начальном этапе изучения React и при программировании простых приложений. Но для более сложных случаев понадобится что-либо посерьезнее...

23.1. Строгий режим

Строгий режим — это инструмент, позволяющий выявить следующие проблемы:

- компонент выводит на странице разное содержимое при повторных вызовах.

  Это происходит, если функция компонента при выполнении получает данные из внешних переменных и эти данные изменяются откуда-либо извне. Подобные компоненты могут вести себя непредсказуемо и создавать проблемы;

- функция, реализующая эффект (см. *разд. 9.1.1*), выдает разные результаты при повторных вызовах.

  Такое случается, если функция, реализующая эффект, производит какое-либо действие, которое перед повторным вызовом эффекта должно быть отменено (например, устанавливает сетевое соединение), но не отменяет его (что должно производиться в функции очистки, возвращаемой этой функцией, — см. *разд. 9.1.1*);

- функция, реализующая функциональный реф (см. *разд. 4.2.1.3*), при повторном вызове выдает другой результат (причиной этого является также отсутствие функции очистки);

- использование устаревших программных инструментов (что может случиться при переделке старого кода или применении старых библиотек).

Чтобы задействовать строгий режим для какого-либо фрагмента интерфейса приложения, надо поместить этот фрагмент в компонент `StrictMode` из модуля `react`:

```
import { StrictMode } from 'react';
. . .
<StrictMode>
    <div>
        <TodoList todos={todos} />
```

```
        <TodoAdd add={add} />
    </div>
</StrictMode>
```

Следует отметить, что компонент `StrictMode` и соответственно строгий режим функционируют лишь при запуске проекта в режиме разработки. При работе в эксплуатационном режиме они отключаются.

Во вновь созданном React-проекте внутрь компонента `StrictMode` помещен сам компонент приложения (см. листинг 3.2, показывающий код стартового модуля), вследствие чего строгий режим распространяется на весь интерфейс веб-приложения без исключения. Однако вы, уважаемые читатели, можете задействовать строгий режим для части интерфейса, убрав компонент `StrictMode` из стартового модуля и вставив его экземпляр в нужную часть кода (как показано в приведенном примере).

> ***За кулисами***
>
> Компонент `StrictMode` производит вывод своего содержимого на экран дважды: монтирует все содержащиеся в нем компоненты, размонтирует их и монтирует снова. При таком двойном выводе проблемы, описанные ранее, проявляются сразу же.

23.2. Настройка отладчика, встроенного в Visual Studio Code

Среда разработки Visual Studio Code содержит мощный отладчик, с помощью которого можно отлаживать React-приложения. Однако этот отладчик требует предварительной настройки, которая выполняется в несколько несложных шагов.

1. Переключитесь на боковую панель **Запуск и отладка**.
2. Щелкните на кнопке **Запуск и отладка** и выберите в появившемся списке один из следующих пунктов:
   - **Веб-приложение (Chrome)** — для отладки создаваемого приложения в веб-обозревателе Google Chrome;
   - **Веб-приложение (Edge)** — для отладки приложения в Microsoft Edge Chromium.

   В результате будет создан файл launch.json, содержащий конфигурацию отладчика, специфическую для рассматриваемого проекта. Он будет сохранен в папке .vscode, находящейся непосредственно в папке проекта.

   Содержимое вновь созданного файла launch.json — JSON-документ с настройками отладчика — будет открыто в отдельной вкладке редактора.
3. Найдите атрибут `configurations` в JSON-документе из файла launch.json.

   Он хранит массив конфигураций отладчика, каждая из которых записана в виде вложенного JSON-документа. Единственный элемент этого массива описывает только что созданную конфигурацию.

   Далее мы будем работать с содержимым этого элемента массива конфигураций.

4. Добавьте атрибут `preLaunchTask` со значением `"npm: start"` (двоеточие после `npm` обязательно).
5. Исправьте записанный в атрибуте `url` интернет-адрес запускаемого приложения, изменив указанный в нем TCP-порт с 8080 на 3000.

> ***Внимание!***
>
> TCP-порт 3000 используется для запуска React-приложения по умолчанию. Если вы изменили его в настройках проекта (они были описаны в *разд. 3.6.3*), в конфигурации отладчика следует указать измененный номер TCP-порта.

6. При желании — измените записанное в атрибуте `name` имя конфигурации на более понятное.
7. Сохраните файл launch.json.

Листинг 23.1 показывает содержимое файла launch.json с конфигурацией, которая должна получиться в результате (комментарии убраны ради компактности).

Листинг 23.1. Файл launch.json с конфигурацией, настраивающей отладчик для запуска React-проекта

```
{
    "version": "0.2.0",
    "configurations": [
        {
            "type": "chrome",
            "request": "launch",
            "name": "React-приложение",
            "url": "http://localhost:3030",
            "webRoot": "${workspaceFolder}",
            "preLaunchTask": "npm: start"
        }
    ]
}
```

23.3. Расширение React Developer Tools

Расширение веб-обозревателей React Developer Tools позволяет просматривать сведения о компонентах, выполнять профилирование и способно помочь в отладке React-приложений. Оно доступно для Google Chrome и Mozilla Firefox, а установить его можно из магазина приложений соответствующего веб-обозревателя.

Расширение React Developer Tools добавляет в окно инструментов разработчика, встроенных в веб-обозреватель, две вкладки, описываемые далее.

23.3.1. Сведения о компонентах

Сведения о компонентах, из которых состоит интерфейс веб-приложения, запущенного на текущей вкладке веб-обозревателя, отображаются на вкладке **Components** (рис. 23.1).

Содержимое этой вкладки делится на две части. Разделяющую их вертикальную линию можно перетаскивать мышью, меняя относительные размеры частей.

В левой части располагается иерархический список компонентов, составляющих интерфейс приложения. Пункты списка отображают имена представляемых ими компонентов. Ветви списка можно сворачивать и вновь разворачивать. По умолчанию все ветви развернуты.

Следует отметить, что в этом списке показаны не только компоненты, непосредственно размещенные разработчиком, но и компоненты, входящие в их состав. Так, на рис. 23.1 в списке присутствуют также компоненты `DataRouter.Provider`, `DataRouterState.Provider`, `Fetchers.Provider`, `ViewTransition.Provider`, `Router` и др., входящие в состав компонентов из библиотеки React Router.

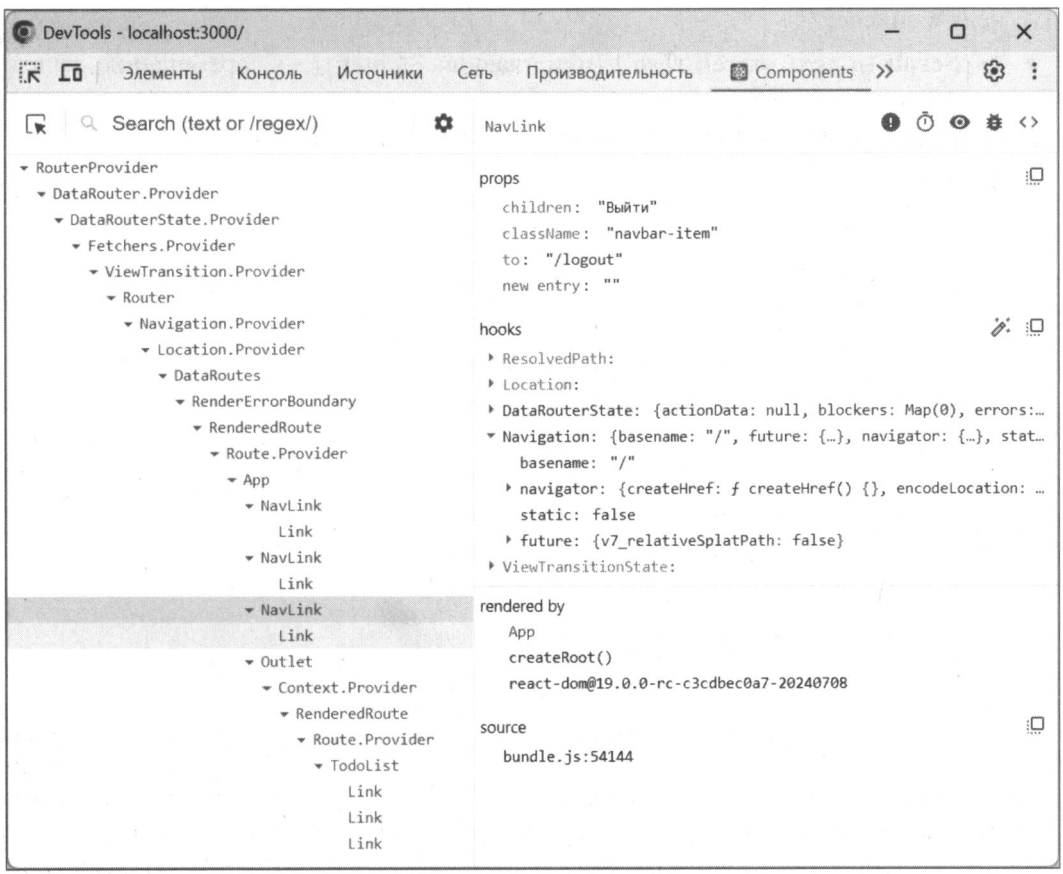


Рис. 23.1. Расширение React Developer Tools, вкладка **Components**

Если навести курсор мыши на какой-либо пункт в списке компонентов, представляемый этим пунктом компонент, располагающийся на странице, будет подсвечен, а рядом отобразятся его имя и геометрические размеры.

Щелкнув мышью на пункте списка, можно выбрать соответствующий компонент для исследования (так, на рис. 23.1 выбран третий по счету компонент **NavLink**).

Можно также щелкнуть на имеющейся в левом верхнем углу вкладки кнопке **Select an element in the page to inspect it**, а потом выбрать нужный компонент непосредственно на странице, просто щелкнув на нем. Чтобы отключить режим выбора компонентов мышью, достаточно еще раз щелкнуть на упомянутой кнопке.

В списке можно выполнять поиск компонентов. Достаточно ввести искомую подстроку в расположенное над списком компонентов поле ввода — и найденные в именах компонентов совпадения будут подсвечены желтым фоном непосредственно в пунктах списка (рис. 23.2).

Правее поля для ввода искомой подстроки находятся следующие кнопки:

- **Scroll to previous search result** (можно также нажать комбинацию клавиш <Shift>+<Enter>) — перемещение выделения на предыдущий найденный компонент в списке;
- **Scroll to next search result** (или клавиша <Enter>) — перемещение выделения на следующий найденный компонент;
- **Reset search** — очистка поля для ввода искомой подстроки.

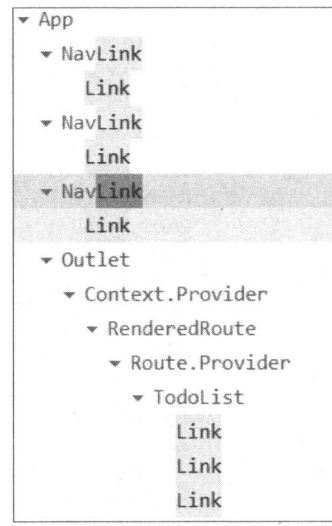


Рис. 23.2. Результат поиска компонентов по подстроке `link`

Левее этих кнопок выводятся порядковый номер выбранного в текущий момент найденного компонента и, через вертикальный разделитель, общее количество найденных компонентов.

В правой части вкладки представлены сведения о компоненте, выбранном в левом списке. Они включают в себя (в порядке сверху вниз):

- имя компонента;
- раздел **props** — пропы, полученные компонентом, вместе с их значениями.

  Значение, представляющее собой массив или объект, можно развернуть и посмотреть его элементы или свойства.

Значения пропов, если они принадлежат к элементарным типам: строковому, числовому или логическому, — можно изменять. Для этого достаточно поставить текстовый курсор на значение, внести нужные правки и нажать клавишу <Enter> или щелкнуть где-либо вне значения. Нажатие клавиши <Esc> отменяет внесенные правки. Значения логического типа также можно менять, устанавливая и сбрасывая флажки, находящиеся правее самих значений.

Под пропами расположен пункт **new entry: ""**, посредством которого можно создать у компонента новый проп. Для этого сначала следует щелкнуть на самой надписи **new entry** и ввести имя создаваемого пропа, а затем щелкнуть на двойных кавычках и указать значение пропа. Строковое значение можно ввести внутри двойных кавычек, значение другого типа — предварительно удалив кавычки.

В правом конце строки с названием раздела **props** находится кнопка  **Copy to clipboard**. Если щелкнуть на ней, значения всех пропов в формате JSON будут скопированы в буфер обмена;

- раздел **hooks** — хуки, используемые в коде компонента. Имена хуков показываются без префикса `use`.

  Правее некоторых хуков выводятся предоставляемые ими значения. Например, правее хука `useState()` выводится текущее значение состояния, правее хука `useMemo()` — текущее мемоизированное значение. У некоторых хуков отображаются возвращенные ими значения (так, на рис. 23.1 у служебного хука `useNavigation()`, используемого компонентами гиперссылок, показано содержимое возвращенного им объекта).

  Текущее значение состояния, относящееся к любому элементарному типу, можно править так же, как и элементарное значение пропа. А текущее значение состояния, представляющее собой массив или объект, можно развернуть, чтобы просмотреть его содержимое.

  Здесь также присутствует кнопка **Copy to clipboard**, по щелчку на которой в буфер обмена копируется JavaScript-код массива простых объектов, каждый из которых представляет один из используемых в компоненте хуков, вместе с возвращенным им результатом и даже хуками, вызываемыми в его коде. Этот код может пригодиться при локализации некоторых ошибок, но разбираться в нем довольно сложно;

- раздел **rendered by** — компонент-родитель, в котором содержится выбранный компонент, и все родители этого родителя;

- раздел **source** — имя модуля и, через двоеточие, номер строки кода, в которой выводится этот компонент.

  По щелчку на кнопке **Copy to clipboard** в буфер обмена будет скопирован полный путь к модулю и, через двоеточие, номер строки кода, в которой выполняется вывод компонента.

В правой части вкладки — в правом конце строки с именем выбранного компонента — присутствуют пять кнопок, позволяющих выполнить с выбранным компонентом различные действия:

- **Force the selected component into an errored state** — имитирует возникновение ошибки в выбранном компоненте. Применяется, чтобы проверить, как приложение обрабатывает ошибки.

  Чтобы вернуть компонент в обычное, «рабочее» состояние, достаточно нажать кнопку **Clear the forced error**, которая появится на месте кнопки **Force the selected component into an errored state** и будет иметь похожее графическое обозначение;

- **Suspend the selected component** — актуальна только у компонентов, загружаемых по запросу средствами React (т. е. с помощью функции `lazy()` и компонента `Suspense`, подробности — в *разд. 22.2*). Нажатие этой кнопки вызывает приостановку загрузки компонента на произвольное время, в течение которого на странице можно наблюдать содержимое, выводимое, когда компонент еще не загружен (задается в пропе `fallback` компонента `Suspense`).

  Чтобы отменить приостановку загрузки компонента и, таким образом, вывести его на странице, следует нажать кнопку **Unsuspend the selected component**, которая появится на месте кнопки **Suspend the selected component** и будет иметь похожее графическое обозначение;

- **Inspect the matching DOM element** — открывает вкладку **Элементы** инструментов разработчика и выбирает в ней корневой тег компонента;

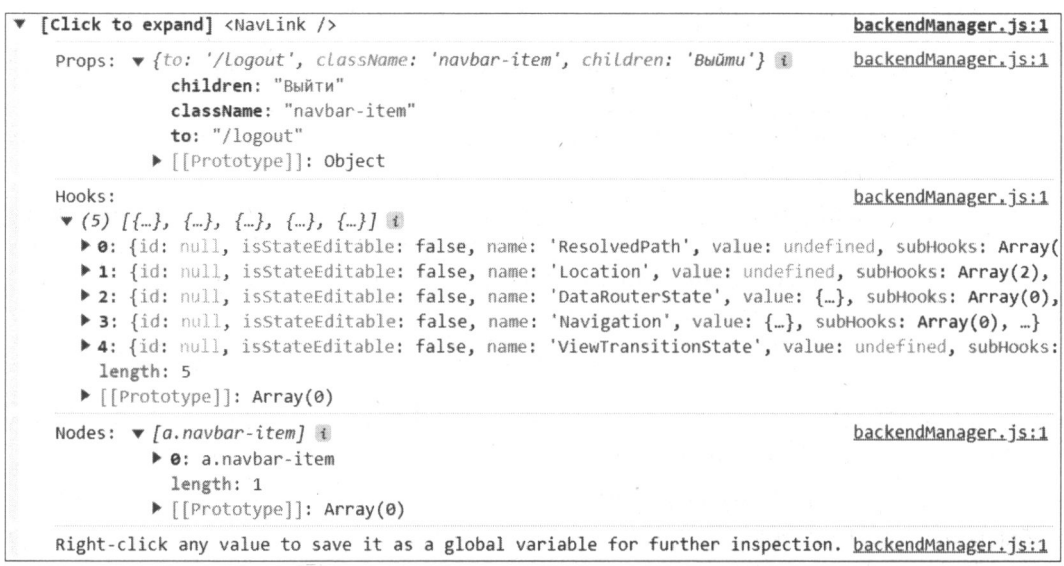


Рис. 23.3. Сведения о компоненте, выведенные в консоли веб-обозревателя

- **Log this component data to the console** — выводит в консоли веб-обозревателя сведения о компоненте в виде иерархического списка (рис. 23.3). Содержит единственную ветвь, на которой отображается тег компонента, включающую следующие вложенные ветви:

- **Props** — перечень всех пропов, указанных у компонента;
- **Hooks** — список всех хуков, вызываемых в коде компонента;
- **Nodes** — список всех элементов страницы, составляющих содержимое компонента;

♦ <> **View source for this component** — открывает вкладку **Источники** инструментов разработчика и выводит в ней модуль с кодом компонента.

23.3.2. Профилировщик

Профилировщик — инструмент, замеряющий время рендеринга как всего интерфейса приложения, так и отдельных компонентов, его составляющих, и представляющий результаты измерения в виде удобного графика. Профилировщик позволяет выяснить, рендеринг какого компонента выполняется слишком долго и (или) часто, и не следует ли мемоизировать этот компонент, чтобы ускорить его вывод.

Профилировщик, встроенный в React Developer Tools, выводится на вкладке **Profiler** (рис. 23.4). Изначально на этой вкладке ничего нет, поскольку по умолчанию профилировщик находится в неактивном состоянии, чтобы не замедлять приложение.

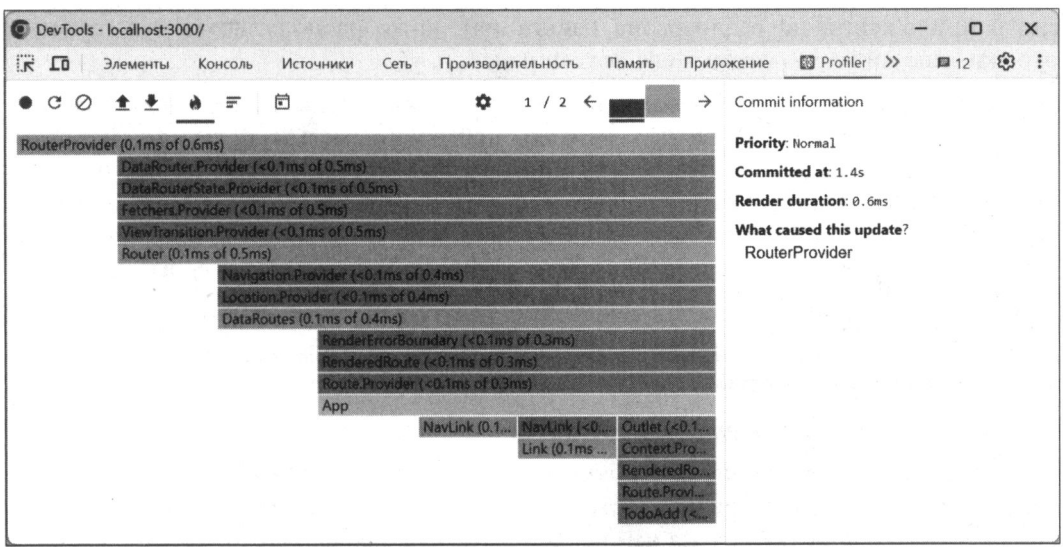


Рис. 23.4. Вкладка **Profiler**, временна́я диаграмма

Чтобы перевести профилировщик в активное состояние, в котором он выполняет все необходимые замеры, следует нажать синюю кнопку ● **Start profiling**, находящуюся в левом верхнем углу вкладки. Эта кнопка тотчас сменится кнопкой **Stop profiling**, похожей на кнопку **Start profiling**, только красного, а не синего цвета. Это является сигналом того, что профилировщик активен.

Пока профилировщик находится в активном состоянии, можно выполнить одно из двух действий:

- выполнить над приложением какие-либо действия, которые приведут к изменению его интерфейса (например, перейти на другой экран);
- обновить страницу приложения, чтобы выполнить повторный рендеринг всего интерфейса.

Правее кнопки **Start profiling** присутствует кнопка **Reload and start profiling**, которая активизирует профилировщик и сразу же обновляет страницу приложения. Поэтому вместо нажатия кнопки **Start profiling** и обновления страницы вручную можно сразу щелкнуть на этой кнопке.

Профилировщик регистрирует каждое обновление интерфейса приложения и замеряет время, в течение которого оно выполняется, сохраняя результаты замеров в так называемом *коммите*. Каждому обновлению интерфейса соответствует отдельный коммит.

Чтобы деактивировать профилировщик и просмотреть результаты измерения времени рендеринга, следует нажать кнопку **Stop profiling**. На вкладке сразу же появится *временна́я диаграмма*, показывающая, в какой момент времени начинается рендеринг того или иного компонента и сколько он продолжается (см. рис. 23.4).

Компоненты в этом графике представляются горизонтальными линиями. Они выстроены в направлении сверху вниз в порядке увеличения уровня вложенности. Самым верхним показан провайдер маршрутизации (или компонент приложения, если в приложении не реализована навигация), ниже располагаются его прямые потомки, еще ниже — потомки его потомков и т. д.

Линии имеют разные цвета, обозначающие продолжительность рендеринга. Синие линии представляют компоненты с наименьшей продолжительностью рендеринга, голубые — выводящиеся чуть дольше, сине-зеленые — еще дольше. Далее по продолжительности рендеринга располагаются темно-желтые и ярко-желтые компоненты. Красные компоненты выводятся дольше всех. Компоненты, вообще не обновлявшиеся при очередном обновлении интерфейса, показываются серым цветом.

Если на линии достаточно места, на ней выводится надпись формата:

<имя компонента> (<продолжительность рендеринга самого компонента> of <продолжительность рендеринга самого компонента и всех его потомков>)

Продолжительность рендеринга измеряется в миллисекундах.

Если навести курсор мыши на любую линию, появится всплывающая подсказка с именем компонента и длительностью его рендеринга. Это может помочь, если линия настолько коротка, что на ней не хватает места для вывода надписи.

Изначально временна́я диаграмма показывает результаты из первого имеющегося коммита. Вывести результаты из остальных коммитов поможет панель коммитов, находящаяся в правом конце верхнего меню вкладки **Profiler** и отображающая список коммитов в горизонтальной ориентации (рис. 23.5).

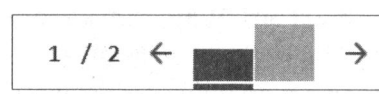


Рис. 23.5. Панель коммитов

Позиции панели представляют отдельные коммиты в виде столбиков разной высоты и разного цвета. Цвет коммита обозначает продолжительность рендеринга при соответствующем обновлении интерфейса приложения.

Чтобы выбрать нужный коммит, достаточно щелкнуть на его столбике. Выбранный коммит помечается горизонтальной черточкой, расположенной под столбиком. Выбирать коммиты также можно, щелкая на кнопках ← **Select previous commit** и → **Select next commit**, расположенных возле столбиков коммитов. В левой части панели коммитов выводится номер выбранного коммита и, через слеш, общее количество коммитов.

Правее панели коммитов, в графе **Commit information**, приводятся сведения о выбранном коммите:

- **Committed at** — время создания коммита в секундах;
- **Render duration** — общая длительность рендеринга в миллисекундах;
- **What caused this update**? — имя компонента, вызвавшего обновление интерфейса приложения.

Если щелкнуть на каком-либо компоненте на временно́й диаграмме, представляющая его линия растянется на всю ширину диаграммы, и соответственно растянутся линии, представляющие потомков этого компонента (рис. 23.6). Таким образом можно временно увеличить масштаб диаграммы, чтобы рассмотреть слишком короткие линии, обозначающие быстро отображающиеся компоненты. Чтобы вернуться к исходному масштабу, следует щелкнуть где-либо на пустом месте в диаграмме.

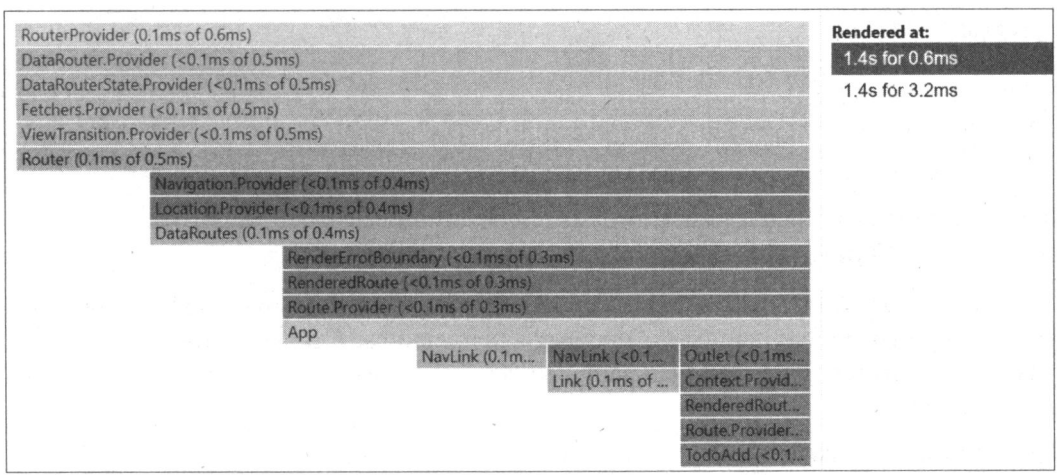


Рис. 23.6. Временна́я диаграмма в увеличенном масштабе (выбран компонент **Router**, создаваемый библиотекой React Router)

Когда на временно́й диаграмме выбран какой-либо компонент, справа появляется графа с именем этого компонента, содержащая список **Rendered at**, который показывает все его коммиты. В этом случае он имеет более привычную вертикальную ориентацию, а его пункты содержат надписи формата:

<общая продолжительность рендеринга> for <продолжительность рендеринга компонента>

Вместо временно́й диаграммы можно вывести *график продолжительности рендеринга*, который наглядно покажет, сколько времени занимает вывод каждого компонента (рис. 23.7). Для этого следует щелкнуть на кнопке **Ranked chart**, находящейся над диаграммой (см. рис. 23.4). Функциональность этого графика аналогична таковой у временно́й диаграммы.

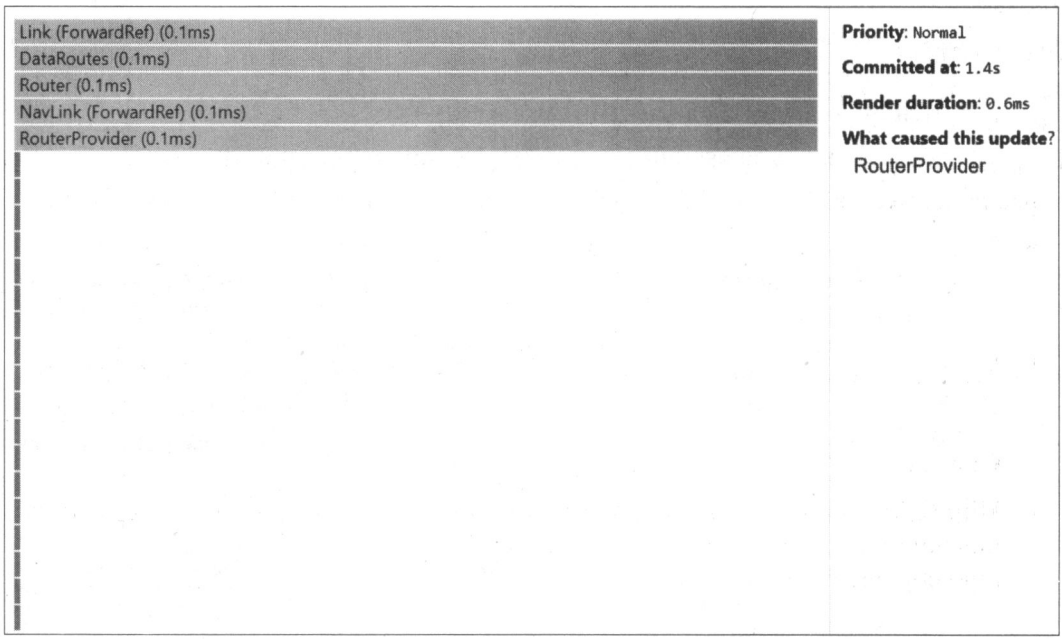


Рис. 23.7. График продолжительности рендеринга

Чтобы вернуться к временно́й диаграмме, достаточно нажать расположенную там же кнопку **Flamegraph chart**.

Просмотрев результаты измерений, можно вновь активизировать профилировщик, чтобы получить новые данные о скорости рендеринга. Старые данные при этом будут потеряны (если не были сохранены в файле — как это сделать, будет рассказано очень скоро).

Над диаграммой также имеются следующие полезные кнопки:

- **Save profile** — сохраняет текущие результаты измерения скорости рендеринга в файле в формате JSON. Созданный файл будет записан веб-обозревателем на локальном диске — либо с запросом местоположения сохраняемого файла, либо без такового (в зависимости от настроек веб-обозревателя);
- **Load profile** — открывает сохраненный ранее файл с результатами измерений. На экране появится стандартное диалоговое окно открытия файла, дающее

возможность выбрать нужный файл. Загруженные результаты будут сразу же выведены на экран;

- ⊘ **Clear profiling data** — удаляет результаты измерений, очищая вкладку.

23.3.3. Настройка расширения React Developer Tools

Вы можете настроить под себя внешний вид и поведение расширения веб-обозревателя React Developer Tools. Далее представлены наиболее полезные его настройки.

Окно настроек вызывается щелчком на кнопке ⚙ **View settings**, которая находится над списком компонентов или над временно́й диаграммой (в зависимости от того, какая вкладка активна). Оно включает четыре вкладки:

- вкладка **General** (рис. 23.8) содержит настройки, затрагивающие интерфейс расширения:
  - **Theme** — раскрывающийся список, задающий цветовую тему: **Auto** (соответствующая системной теме, выбрана по умолчанию), **Light** (светлая) или **Dark** (темная);
  - **Display density** — раскрывающийся список, задающий размеры пунктов списка компонентов, линий временно́й диаграммы и графика продолжительности рендеринга: **Compact** (малый размер, выбран по умолчанию) или **Comfortable** (увеличенный размер);
  - **Highlight updates when components render** — флажок, включающий подсвечивание компонентов, которые в текущий момент обновляются (по умолчанию сброшен).

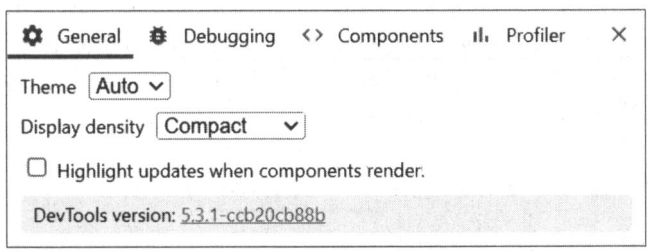


Рис. 23.8. Окно настроек React Developer Tools, вкладка **General**

В ряде случаев стоит установить этот флажок, чтобы знать, обновляется ли тот или иной компонент;

- на вкладке **Debugging** (рис. 23.9) можно указать параметры вывода сообщений об ошибках в консоли веб-обозревателя, используя следующие флажки:
  - **Append component stacks to console warnings and errors** — включает вывод в сообщениях об ошибках стеков вызова функций, реализующих компоненты. Стеки вызова выводятся свернутыми, и для просмотра их следует развернуть. Установлен по умолчанию;

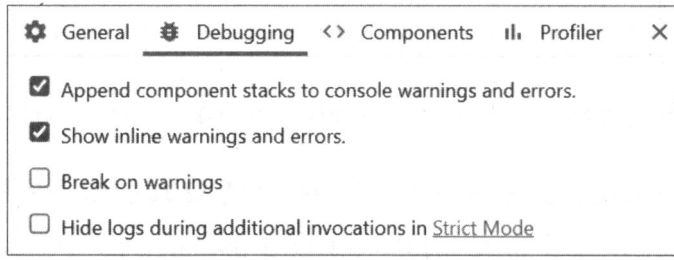


Рис. 23.9. Окно настроек React Developer Tools, вкладка **Debugging**

- **Show inline warnings and errors** — включает вывод в составе сведений о компонентах (см. рис. 23.1) пометок, обозначающих компоненты, при выводе которых возникли ошибки. Такой же пометкой отмечается компонент, у которого была включена имитация ошибки (нажатием кнопки **Force the selected component into an errored state**, подробнее — в *разд. 23.3.1*). Установлен по умолчанию;
- **Break on warnings** — включает прерывание работы приложения при некритических ошибках. Вовремя выявив и исправив такого рода ошибку, можно предотвратить появление других ошибок, на этот раз более критических (по крайней мере, в теории). По умолчанию сброшен;
- **Hide logs during additional invocations in Strict Mode** — включает подавление вывода дублирующихся сообщений об ошибках при повторном выводе компонентов в строгом режиме (см. *разд. 23.1*). По умолчанию сброшен;

♦ вкладка **Components** (рис. 23.10) содержит настройки, касающиеся сведений о компонентах:

- **Expand component tree by default** — флажок, включающий автоматическое разворачивание всех ветвей списка компонентов (по умолчанию установлен);
- **Always parse hook names from source** — флажок, включающий извлечение имен хуков, используемых в выбранном компоненте, из исходного кода (по умолчанию сброшен).

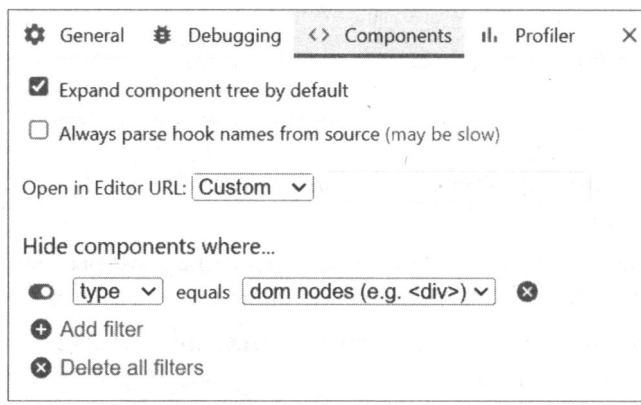


Рис. 23.10. Окно настроек React Developer Tools, вкладка **Components**

Желательно включить этот флажок (если, конечно, это не очень замедлит приложение);

- группа элементов управления **Hide components where** позволяет указать компоненты, которые *не* будут выводиться в иерархическом списке. Скрываемые компоненты должны удовлетворять одному из условий (или фильтров), набор которых задается в этой группе. Условий в наборе может быть произвольное количество;
- переключатель — позволяет временно отключить текущее условие, не удаляя его;
- раскрывающийся список type — задает характеристику компонента. Если эта характеристика совпадет с заданным сравниваемым значением, компонент будет скрыт. Доступны для выбора следующие пункты:
  - **name** — имя компонента.

    Сравниваемое имя в виде регулярного выражения, представленного строкой, без ограничивающих символов `/`, заносится в расположенное правее поле ввода **matches**. Например, если ввести регулярное выражение `.*Link`, будут скрыты компоненты `Link` и `NavLink`;
  - **type** — разновидность компонента.

    Сравниваемая разновидность выбирается в расположенном правее раскрывающемся списке **equals**. В нем доступны для выбора следующие пункты:
  - **context** — провайдер контекста;
  - **dom nodes (e.g.)** — обычный элемент страницы, создаваемый HTML-тегом;
  - **memo** — мемоизированный компонент;
  - **profiler** — компонент-профилировщик (будет описан далее);
  - **suspense** — компонент `Suspense`;
  - **hoc** — выводить только компоненты высшего порядка (см. *главу 10*).
- кнопка **Add filter** добавляет в набор новое условие;
- кнопка **Delete filter**, находящаяся в составе условия, удаляет его.

  По умолчанию существует единственное условие, предписывающее не выводить в списке обычные HTML-теги;
- кнопка **Delete all filters**, расположенная в самом низу, удаляет все условия;

♦ вкладка **Profiler** предоставляет настройки профилировщика и содержит два флажка:

- **Record why each component rendered while profiling** — включает регистрацию события, которое привело к обновлению того или иного компонента: обновление состояния, значений пропов и т. п. (по умолчанию сброшен).

  Этот флажок желательно установить сразу же и впоследствии не сбрасывать;

- **Hide commits below** — включает скрытие коммитов с общим временем рендеринга менее заданного в поле ввода, расположенном правее. Время указывается в миллисекундах.

Все заданные настройки применяются сразу же, без необходимости нажимать какие-либо кнопки наподобие **ОК**.

Закрыть окно настроек можно щелчком на кнопке закрытия, расположенной в его правом верхнем углу и имеющей вид крестика, или щелчком где-либо вне этого окна.

23.3.4. Упрощение отладки собственных хуков

При отладке собственных хуков есть возможность вывести в составе сведений об отлаживаемом хуке (в разделе **hooks** правой части вкладки **Components**, описанной в *разд. 23.3.1*) какое-либо значение, вычисленное в его теле, или даже несколько таких значений. Это может значительно упростить отладку.

Вывести заданное *значение* позволит хук `useDebugValue()` из модуля `react`:

```
useDebugValue(<выводимое значение>[, <форматирующая функция>=undefined])
```

Если *форматирующая функция*:

- *не* указана — заданное *выводимое значение* будет выведено без каких-либо преобразований;
- указана — она будет вызвана с передачей ей *выводимого значения* в качестве единственного параметра, и в разделе **hooks** сведений о компоненте отобразится результат, возвращенный *функцией*.

Листинг 23.2 содержит код хука `useShowNavMenu()`, предназначенного для управления выводом меню навигации. В коде присутствуют вызовы хука `useDebugValue()`, выводящие соответственно надпись «Выведено» или «Скрыто» и временну́ю отметку вывода или скрытия меню.

Листинг 23.2. Код хука useShowNavMenu()

```
import { useState, useDebugValue } from 'react';
export function useShowNavMenu(initialState) {
    const [visible, setVisible] = useState(initialState);
    const toggleVisible = () => {setVisible(!visible)};
    useDebugValue(visible ? 'Выведено' : 'Скрыто');
    useDebugValue(new Date(), date => date.toLocaleString());
    return [visible, toggleVisible];
}
```

Применение этого хука:

```
import { useShowNavMenu } from './hooks.js';

export default function App() {
    const [menu, toggleMenu] = useShowNavMenu(false);

    const handleBurgerClick = evt => {
        evt.preventDefault();
        toggleMenu();
    };
    . . .
    return (
        . . .
        {menu && <nav class="nav-menu">. . .</nav>}
        . . .
    );
}
```

В разделе **hooks** правой части вкладки **Components** будут выведены значения, заданные в вызовах хука `useDebugValue()` (рис. 23.11).

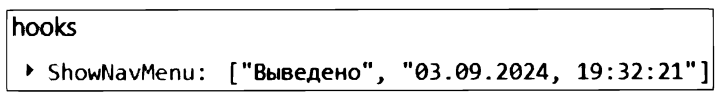


Рис. 23.11. Значения, выведенные в разделе hooks вкладки Components с помощью хука `useDebugValue()`

23.4. Расширение Redux DevTools

Расширение веб-обозревателей Redux DevTools позволяет просматривать текущее состояние хранилища Redux, список влияний, отправленных этому хранилищу, и изменения, которые эти влияния вносят в состояние хранилища. Оно доступно для Google Chrome и Mozilla Firefox, а установить его можно из магазина приложений соответствующего веб-обозревателя.

23.4.1. Подготовка проекта к работе с Redux DevTools

Чтобы расширение Redux DevTools смогло получать сведения о работе хранилища из отлаживаемого приложения, в код приложения следует внести некоторые правки. А именно: при создании хранилища указать третьим параметром функции `createStore()` (см. *разд. 14.1.3*) расширитель, возвращаемый методом `__REDUX_DEVTOOLS_EXTENSION__()` объекта `window`. Этот метод объявляется самим расширением. Пример:

```
const store = createStore(todosReducer, initialData,
                          window.__REDUX_DEVTOOLS_EXTENSION__());
```

Если к хранилищу требуется привязать еще один расширитель (например, составляемый из посредников вызовом функции `applyMiddleware()`, подробности — в *разд. 14.2.1*), следует передать методу `__REDUX_DEVTOOLS_EXTENSION_COMPOSE__()` объекта `window` (этот метод также объявляется расширением) другой привязываемый расширитель, а результат, возвращенный этим методом, передать третьим параметром функции `createStore()`. Пример:

```
const store = createStore(todosReducer, initialData,
    window.__REDUX_DEVTOOLS_EXTENSION_COMPOSE__(
        applyMiddleware(
            userChangedMiddleware,
            loadTodosMiddleware,
            . . .
        )
    )
);
```

Если при разработке применялась библиотека Redux Toolkit (см. *разд. 15.2*), никаких правок в код вносить не нужно, т. к. при создании хранилища средствами этой библиотеки к хранилищу привязывается расширитель, поставляемый в составе библиотеки Redux DevTools Extension (которая устанавливается вместе с Redux Toolkit) и обеспечивающий работу с расширением.

23.4.2. Работа с Redux DevTools

Расширение Redux DevTools выводится на вкладке **Redux** окна инструментов разработчика, встроенных в веб-обозреватель (рис. 23.12). Рассмотрим наиболее полезные инструменты, предлагаемые этим расширением.

Прежде всего, следует знать, что расширение создает копию состояния хранилища, перехватывает все влияния, отправляемые хранилищу, и вносит все задаваемые ими изменения в созданную копию. Кроме того, оно позволяет откатиться на выбранное влияние и посмотреть, какие данные находились в хранилище на момент его обработки. То есть ведет своего рода летопись состояния хранилища, что может пригодиться в ряде случаев.

Расширение способно отображать процесс функционирования хранилища в трех разных режимах. Эти режимы выбираются щелчками на кнопках, находящихся в центре нижнего края вкладки (см. рис. 23.12). Кнопка, соответствующая выбранному в текущий момент режиму, выделяется подчеркиванием.

Начнем с рассмотрения режима **Inspector** — по мнению автора, наиболее удобном (и задействованном по умолчанию — на рис. 23.12 как раз показан этот режим в действии).

В левой части вкладки выводится список всех влияний, отправленных хранилищу и перехваченных расширением. Влияния выводятся сверху вниз в порядке от отправленных ранее к отправленным позднее.

Первый пункт списка в левой части содержит надпись **@@INIT** и обозначает момент создания хранилища. В правой части пункта показывается время создания

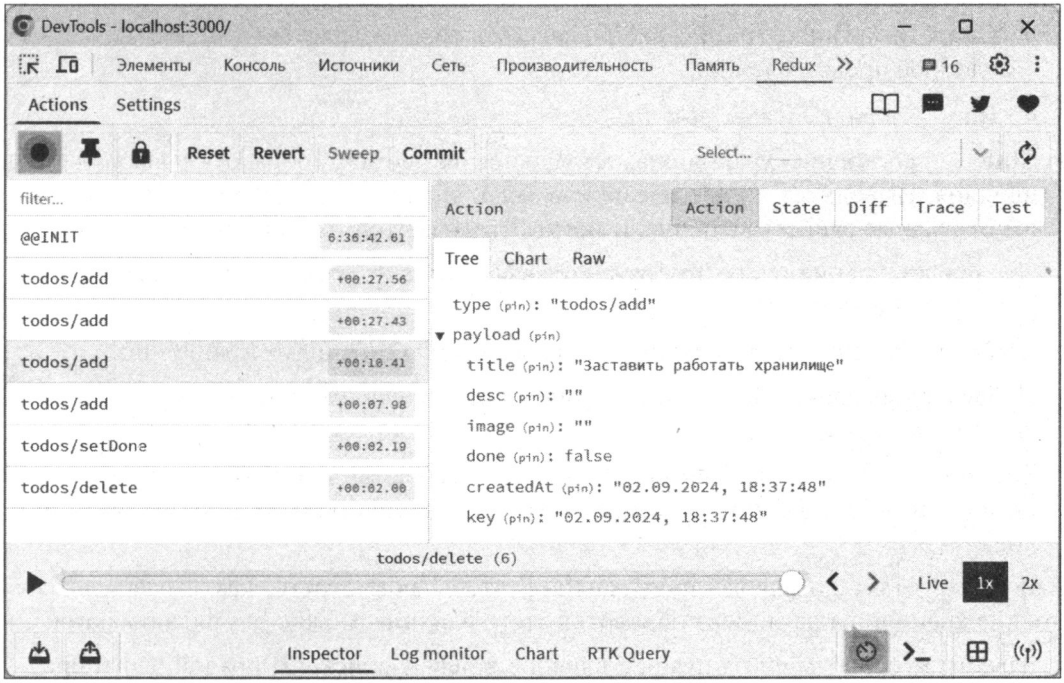


Рис. 23.12. Расширение Redux DevTools, режим **Inspector**

хранилища в 12-часовом формате (переключить его в 24-часовой формат, к сожалению, нельзя).

Каждый следующий пункт списка представляет одно влияние. В его левой части выводится тип влияния, а в правой — время отправки, отсчитанное от времени создания хранилища.

Выводящиеся в списке влияния можно фильтровать. Достаточно ввести в расположенное над списком поле ввода **filter** искомую подстроку — и в списке будут выведены только те влияния, в наименовании типа которых присутствует заданная подстрока.

Любое влияние в списке можно выбрать и просмотреть сведения о нем в правой части вкладки. Эти сведения делятся на несколько видов, переключаться между которыми можно, щелкая на кнопках, которые располагаются над выводимыми сведениями:

- **Action** — содержимое влияния, а именно свойства представляющего его объекта (`type`, `payload` и др.) вместе со значениями. Это содержимое выводится в блокноте с тремя вкладками, каждая из которых отображает содержимое в разных форматах:
  - **Tree** — в виде иерархического списка свойств и их значений.

    Значения, являющиеся массивами и объектами, можно разворачивать и просматривать их содержимое (так, на рис. 23.12 развернуто значение свойства `payload` влияния);

- **Chart** — в виде графической диаграммы (на взгляд автора, это не самое удобное представление);
- **Raw** — в виде JavaScript-кода;

♦ **State** — состояние хранилища, актуальное на момент обработки выбранного влияния. Выводится в виде иерархического списка, ветви которого представляют отдельные свойства влияния и могут быть развернуты и свернуты.

Это представление также отображается в блокноте из трех вкладок, аналогичном рассмотренному ранее;

♦ **Diff** — различия между текущим и предыдущим состояниями хранилища.

Добавленные данные выделяются темно-зеленым фоном, удаленные — красным фоном и зачеркиванием.

Различия между состояниями хранилища отображаются в блокноте из двух вкладок и доступны для просмотра в виде иерархического списка и JavaScript-кода.

Выполнить откат до какого-либо влияния, чтобы посмотреть, как выглядело состояние хранилища на момент обработки этого влияния, можно двумя способами:

♦ навести курсор мыши на нужное влияние в левом списке. В правой части представляющего его пункта вместо времени отправки появятся две кнопки. Откат будет выполнен по щелчку на кнопке **Jump** (вторую кнопку — **Skip** — мы рассмотрим чуть позже);

♦ сдвинуть влево на соответствующую позицию ползунок регулятора, находящегося в нижней части вкладки.

Также можно переходить на предыдущее или последующее влияния, щелкая на расположенных правее регулятора кнопках со стрелками соответственно влево и вправо.

Вернуться к состоянию хранилища, актуальному на момент получения последнего влияния, можно также двумя способами:

♦ навести курсор мыши на самое последнее влияние в левом списке и щелкнуть на появившейся кнопке **Jump**;

♦ сдвинуть ползунок регулятора в крайнюю правую позицию.

Можно также щелкнуть на кнопке воспроизведения, находящейся левее регулятора, и наблюдать, как хранилище постепенно, от влияния к влиянию, возвращается к актуальному состоянию. Можно приостановить воспроизведение, нажав на кнопку паузы (она появится на месте кнопки воспроизведения), а потом возобновить его.

Еще есть возможность регулировать скорость движения ползунка регулятора. Для этого служат две расположенные правее регулятора кнопки: **1x** (обычная скорость «воспроизведения») и **2x** (удвоенная скорость)

Можно временно отменить изменения, внесенные в хранилище произвольным влиянием. Для этого достаточно навести курсор мыши на нужный пункт левого

списка и нажать на появившуюся кнопку **Skip**. Вновь ввести влияние «в строй» можно, щелкнув на этой кнопке еще раз.

Хранение списка влияний, полученных хранилищем, и состояний хранилища на момент получения каждого влияния отнимает много оперативной памяти. Поэтому время от времени рекомендуется выполнять «подведение итогов», при котором изменения в состоянии хранилища, создаваемые всеми полученными влияниями, непосредственно переносятся в хранилище, а список влияний очищается. Это выполняется нажатием кнопки **Commit**, которая находится над полем ввода **filter**, справа.

Помимо кнопки **Commit**, над этим полем ввода находятся три другие кнопки:

- **Reset** — возврат хранилища к изначальному состоянию;
- **Revert** — возврат хранилища к состоянию, актуальному на момент возобновления отслеживания влияний (как приостановить и возобновить отслеживание влияний, рассказано далее);
- **Sweep** — удаление из списка всех отмененных влияний (у которых была нажата кнопка **Skip**).

В режиме **Log monitor** (рис. 23.13) почти всю вкладку занимает список полученных хранилищем влияний, в каждом пункте которого выводятся тип очередного влияния, данные, передающиеся в этом влиянии, и состояние хранилища после обработки влияния. Данные влияния и состояние хранилища представляются иерархическими списками.

При переключении в режим **Chart** на вкладке выводится малополезная диаграмма, показывающая текущее состояние хранилища.

Над полем ввода **filter**, слева, расположены следующие полезные кнопки:

-  **Pause recording** — временная приостановка отслеживания получаемых влияний.

  Может пригодиться, чтобы пропустить отслеживание гарантированно работающих операций с хранилищем, тем самым сохранив память и повысив производительность приложения;
-  **Start recording** — возобновление отслеживания получаемых влияний после приостановки. Выводится вместо кнопки **Pause recording**;
-  **Lock changes** — временное блокирование любых операций с хранилищем.

  В результате данные, содержащиеся в хранилище, при работе с приложением изменяться не будут;
-  **Unlock changes** — разблокирование хранилища после его блокировки, выводится вместо кнопки **Lock changes**.

В правом нижнем углу вкладки присутствуют следующие кнопки, также могущие оказаться полезными:

-  **Hide slider** — скрывает панель с регулятором, посредством которого можно выполнить откат до заданного влияния;

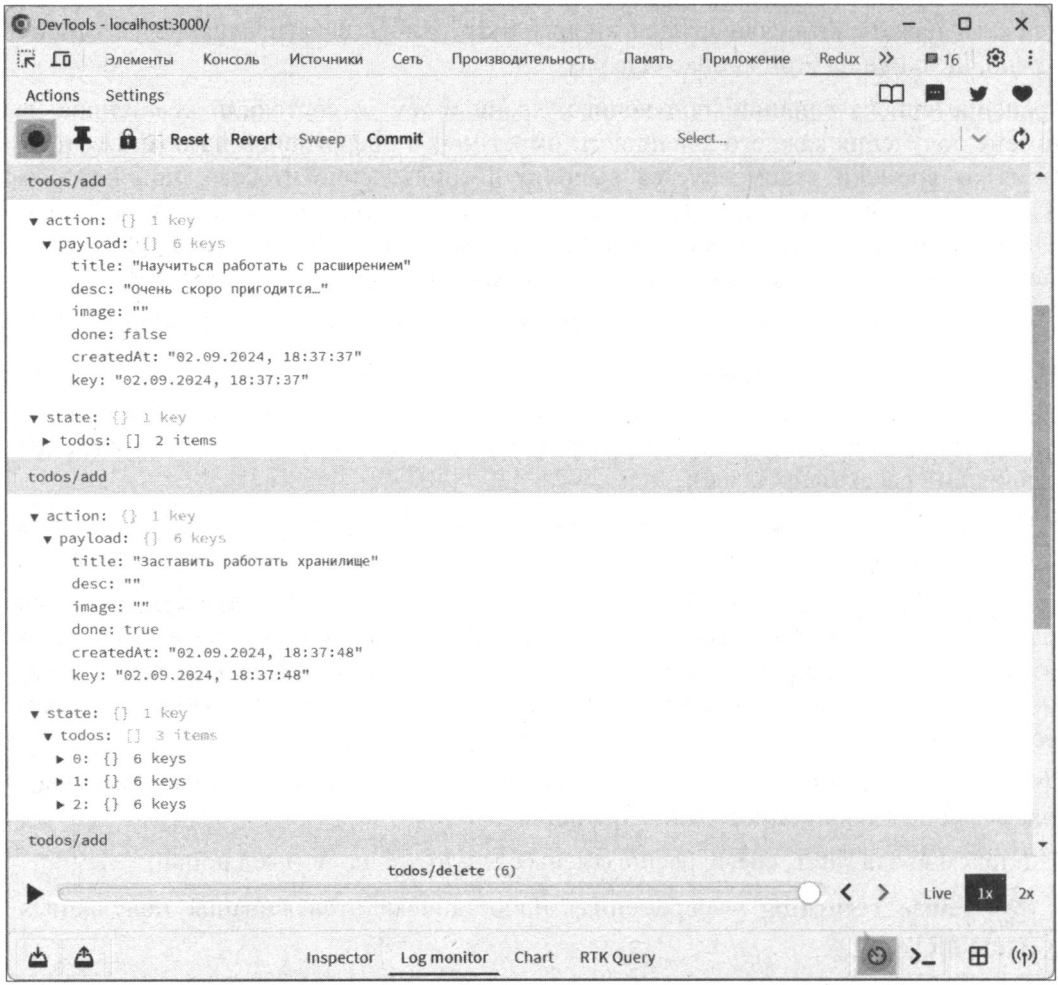


Рис. 23.13. Расширение Redux DevTools, режим **Log monitor**

- **Show slider** — вновь выводит панель с регулятором, скрытую нажатием кнопки **Show slider**;
- **Show dispatcher** — выводит внизу вкладки панель с областью редактирования, в которое можно ввести произвольное влияние и отправить его хранилищу нажатием расположенной в той же панели кнопки **Dispatch**. Код влияния вводится на языке JavaScript в виде простого объекта;
- **Hide dispatcher** — скрывает панель для ввода собственных влияний, выведенную нажатием кнопки **Show dispatcher**.

Осталось рассмотреть две кнопки, находящиеся в левом нижнем углу вкладки:

- **Export** — сохраняет текущее состояние хранилища в файле в формате JSON. Созданный файл будет записан веб-обозревателем на локальном диске —

либо с запросом местоположения сохраняемого файла, либо без такового (в зависимости от настроек веб-обозревателя);

- **Import** — загружает сохраненный ранее файл с состоянием хранилища. На экране появится стандартное диалоговое окно открытия файла, в котором можно выбрать нужный файл.

23.4.3. Настройка расширения Redux DevTools

Настройки Redux DevTools можно задать, нажав на кнопку **Settings**, расположенную в левом верхнем углу вкладки **Redux**, выше кнопок **Pause recording** и **Lock changes**. Откроется страница настроек с двумя вкладками: **Themes** и **State Tree**.

Наиболее полезна вкладка **Themes**, содержащая три раскрывающихся списка:

- **theme** — тема оформления вкладки **Redux**: **default** (легкая, выбрана по умолчанию) или **material** (более тяжеловесная);
- **color scheme** — цветовая схема вкладки. Для выбора доступно очень много — несколько десятков — схем. По умолчанию: **default**;
- **theme color** — цветовая тема вкладки: **auto** (совпадает с системной темой, выбрана по умолчанию), **light** (светлая) или **dark** (темная).

Все заданные настройки применяются сразу же, без необходимости нажимать какие-либо кнопки наподобие **OK**.

Чтобы вернуться к странице сведений о хранилище и влияниях, описанной в *разд. 23.4.2*, достаточно нажать кнопку **Actions**, находящуюся в левом верхнем углу вкладки, левее кнопки **Settings**.

23.5. Получение метрик Web Vitals

Метрики производительности Web Vitals — единая система измерения производительности веб-приложений, предложенная корпорацией Google.

В коде стартового модуля src\index.js присутствует вызов функции `reportWebVitals()` из модуля src\reportWebVitals.js. Эта функция вычисляет метрики производительности, но изначально никуда их не выводит. Чтобы вывести вычисленные метрики в консоль веб-обозревателя, следует передать функции `reportWebVitals()` в качестве параметра ссылку на метод `log()` объекта `console`, например:

```
reportWebVitals(console.log);
```

23.6. Встроенный профилировщик React

Помимо профилировщика, являющегося частью расширения React Developer Tools, для измерения продолжительности рендеринга можно использовать также профилировщик, встроенный в сам React.

Профилировщик React реализуется компонентом `Profiler` из модуля `react`. Он должен включать в себя фрагмент содержимого, у которого следует измерить продолжительность рендеринга.

Компонент `Profiler` поддерживает следующие пропы:

- `id` — уникальный идентификатор текущего экземпляра профилировщика. Задается в виде строки и может быть выбран произвольно;
- `onRender` — функция, вызываемая после рендеринга содержимого профилировщика и, собственно, обрабатывающая результаты измерений. Она должна принимать следующие параметры:
  - идентификатор экземпляра профилировщика, заданный в пропе `id`;
  - обозначение этапа жизненного цикла компонента в виде строки: `'mount'` (монтирование), `'update'` (обновление) или `'nested-update'` (обновление, только в планировщиках, вложенных в другой планировщик). Жизненный цикл компонента и его этапы описаны в *разд. 4.1.1*;
  - продолжительность рендеринга содержимого профилировщика в наиболее оптимальном режиме в виде целого числа в миллисекундах.

    В наилучшем случае продолжительность рендеринга при обновлении компонента должна быть существенно меньше продолжительности рендеринга при монтировании. Если это не так, возможно, следует подумать о мемоизации содержимого или его части;
  - продолжительность рендеринга содержимого профилировщика без использования каких бы то ни было оптимизаций, которые обеспечивает React, в виде целого числа в миллисекундах.

    Крайне желательно, чтобы значение предыдущего параметра было существенно меньше значения этого параметра;
  - время начала рендеринга в виде числа миллисекунд, прошедших после запуска приложения;
  - время окончания рендеринга в виде числа миллисекунд, прошедших после запуска приложения.

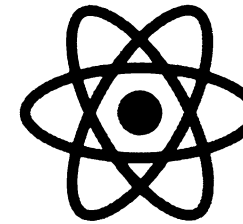
Пример:

```
import { Profiler } from 'react';

export default function App() {
    . . .
    const showProfilerResults = (id, phase, fullDuration, contentsDuration,
                                 startTime, endTime) => {
        console.log(`Профилировщик: ${id}`);
        console.log(`${phase === 'mount' ? 'Монтирование' : 'Обновление'}`);
        console.log(`Продолжительность рендеринга с оптимизацией: ` +
                    `${fullDuration}`);
```

```
        console.log(`Продолжительность рендеринга без оптимизации: ` +
                    `${contentsDuration}`);
        console.log(`Начало рендеринга: ${startTime}`);
        console.log(`Окончание рендеринга: ${endTime}`);
    };
    return (
        <Profiler id="MainProfiler" onRender={showProfilerResults}>
            <div className="container">
                . . .
            </div>
        </Profiler>
    );
}
```

ГЛАВА 24



Публикация веб-приложения

Веб-приложение, написанное на React, может быть опубликовано на мощностях как сторонних хостинг-провайдеров, так и самого разработчика. В последнем случае для публикации можно использовать популярный веб-сервер Apache HTTP Server.

Однако перед публикацией следует создать эксплуатационную редакцию приложения. А перед этим — произвести над ним некоторые подготовительные действия.

24.1. Подготовка веб-приложения к публикации

Процесс подготовки приложения к публикации включает:

- удаление выражений, импортирующих ненужные сущности (переменные, функции, классы), как из модулей, написанных самим разработчиком, так и из дополнительных библиотек.

  Если этого не сделать, код, импортирующий ненужные сущности, равно как и код, объявляющий их, будет включен в состав чанков эксплуатационной редакции, увеличит их размер и замедлит запуск приложения;

- удаление кода, служащего лишь для целей отладки. К такого рода коду относятся выражения, выводящие какие-либо значения в консоль веб-обозревателя, компоненты-профилировщики `Profiler`, код, связывающий приложение с отладочными расширениями (наподобие Redux DevTools), и т. п.

  Оставленный в приложении, такой код увеличит объем эксплуатационной редакции и может вызвать различные побочные эффекты (наподобие вывода сообщений в консоль), которые могут обескуражить пользователя;

- настройка параметров проекта, оказывающих влияние на эксплуатационную редакцию приложения (они были описаны в *разд. 3.6.3*).

24.2. Создание эксплуатационной редакции веб-приложения

Отладочная редакция приложения, создаваемая сборщиком Webpack при запуске проекта, хранится исключительно в оперативной памяти. Это позволяет ускорить запуск приложения и исключить появление на локальном диске большого количества файлов, хранящих разные версии кода приложения, которые будут генерироваться при каждом его запуске.

Кроме того, отладочная редакция приложения содержит много кода, необходимого лишь для собственно отладки, а также неиспользуемый библиотечный код и комментарии. Все это приводит к увеличению объема кода.

Поэтому для публикации следует использовать специально создаваемую эксплуатационную редакцию приложения, которая *не* содержит:

- отладочного кода;
- неиспользуемого библиотечного кода;
- комментариев (которые у иных разработчиков могут занимать килобайты);
- ненужных пробелов и разрывов строк.

В результате чего эксплуатационная редакция кода окажется заметно компактнее отладочной, что положительно скажется как на скорости загрузки и запуска, так и на быстродействии приложения.

Для создания эксплуатационной редакции следует в командной консоли перейти в папку проекта и подать команду:

```
npm run build
```

По умолчанию созданная этой командой эксплуатационная редакция приложения сохраняется в папке build, находящейся в папке проекта. Можно выполнить ее сохранение в другую папку, указав новый путь в настройке проекта `BUILD_PATH` (описание всех поддерживаемых настроек проекта приведено в *разд. 3.6.3*).

Эксплуатационная редакция включает следующие папки и файлы:

- index.html — страница приложения;
- все файлы из папки public проекта, привязанные к странице приложения (manifest.json, favicon.ico, logo192.png, logo512.png и др.);
- robots.txt — файл с указаниями для поисковых служб;
- asset-manifest.json — файл с перечнем всех чанков, хранящих код приложения. Он необходим для работы кода, выполняющего загрузку отдельных чанков и генерируемого Webpack;
- static — папка, хранящая таблицы стилей и программный код приложения. Содержит следующие папки:
  - css — папка с CSS-чанками:
    - main.*<хеш>*.chunk.css — весь CSS-код, как написанный разработчиком приложения, так и библиотечный;

- main.<*хеш*>.map — карта исходного кода для CSS-чанка. Создается, только если у настройки проекта GENERATE_SOURCEMAP было указано значение true;

- js — папка с JavaScript-чанками:
  - main.<*хеш*>.chunk.js — код, написанный разработчиком приложения, и библиотечный код (включая код самого React);
  - <*произвольный номер*>.<*хеш*>.chunk.js — чанки, загружаемые по запросу как в самом приложении, так и в используемых им библиотеках. Таких чанков может быть произвольное количество.

    Если в коде приложения и использованных им библиотек не выполняется загрузка кода по запросу, подобного рода файлы не создаются;
  - runtime-main.<*хеш*>.chunk.js — код, управляющий загрузкой связки и генерируемый Webpack. Создается в случае, если настройке проекта INLINE_RUNTIME_CHUNK дано значение false;
  - <*имя чанка*>.map — карта исходного кода для JavaScript-чанка с указанным *именем*. Создается, только если у настройки проекта GENERATE_SOURCEMAP было указано значение true;
  - main.<*хеш*>.chunk.js.LICENSE.txt — сведения о лицензиях, по которым распространяются библиотеки, используемые в проекте, включая сам React;

- media — папка с файлами из папки src, не являющимися таблицами стилей или веб-сценариями, которые используются в приложении (в их число входят изображения, аудио-, видеофайлы и др.). Все файлы в этой папке имеют имена формата:

  <*изначальное имя без расширения*>.<*хеш*>.<*изначальное расширение*>

Папка build уже занесена в список исключений для Git, хранящийся в файле .gitignore. Так что даже после создания эксплуатационной редакции можно продолжать работу над приложением, не опасаясь, что в коммиты Git попадут нежелательные файлы. Если же для сохранения эксплуатационной редакции указана другая папка, следует внести ее в список исключений вручную.

24.3. Публикация веб-приложения

24.3.1. Публикация на мощностях разработчика

24.3.1.1. Публикация с помощью Apache HTTP Server

Apache HTTP Server — наиболее популярный в настоящее время веб-сервер. Поэтому многие разработчики пользуются именно им.

♦ Если навигация в приложении реализуется посредством маршрутизатора Hash Router (описан в *разд. 11.2.4.2*) — файлы эксплуатационной редакции приложения достаточно просто поместить в корневую папку веб-сервера.

  Автор напоминает, что эксплуатационная редакция сохраняется в папке build папки проекта или в папке, путь которой был указан в настройках проекта.

- Если навигация реализована с помощью маршрутизатора Browser Router (см. *разд. 11.2.4.1*) — в корневую папку сервера, помимо самого приложения, следует поместить файл .htaccess с содержимым, представленным в листинге 24.1 (этот файл также можно найти в папке apache-http-server-config сопровождающего книгу электронного архива — см. *приложение* 3).

Листинг 24.1. Файл .htaccess

```
Options -MultiViews
RewriteEngine On
RewriteCond %{REQUEST_FILENAME} !-f
RewriteRule ^ index.html [QSA,L]
```

Файл .htaccess содержит указания для Apache HTTP Server при получении запроса с любым путем отправлять веб-обозревателю страницу приложения. Благодаря этому приложение будет успешно работать при попытке перейти на какой-либо его экран путем прямого ввода интернет-адреса этого экрана.

24.3.1.2. Публикация с помощью веб-сервера serve

serve — это компактный веб-сервер, написанный на JavaScript и работающий под управлением Node.js. Он неплохо подходит для публикации приложений с невысокой нагрузкой.

serve устанавливается на глобальном уровне подачей команды:

```
npm install serve -g
```

> ***На заметку***
>
> Полное описание serve приведено по интернет-адресу: **https://github.com/vercel/serve**.

Чтобы опубликовать веб-приложение с помощью этого веб-сервера, следует в командной консоли перейти в папку, где находится эксплуатационная редакция публикуемого приложения, и запустить сервер следующей командой:

```
serve --single
```

По умолчанию приложение становится доступным через TCP-порт 3000. Следовательно, открыть его можно, перейдя по интернет-адресу **http://localhost:3000/**.

Для остановки этого веб-сервера достаточно переключиться в экземпляр командной консоли, в котором он был запущен, и нажать комбинацию клавиш <Ctrl>+<C>.

Веб-сервер serve поддерживает следующие полезные параметры и командные ключи:

- *<путь к папке с публикуемым приложением>* — *путь* можно указать как абсолютный:

```
serve d:\work\todos\build --single
```

так и относительно текущей папки:

```
d:
cd work\todos
serve build --single
```

Если *путь* не указан, веб-сервер будет обрабатывать содержимое текущей папки;

- `--listen` <*номер TCP-порта*> (вместо ключа `--listen` можно указать сокращение `-l`) — задает TCP-порт, через который будет доступно приложение:

```
serve d:\work\todos\build --listen 80 --single
```

Если ключ не указан, веб-приложение будет доступно через TCP-порт 3000;

- `--single` (или `-s`) — указывает при получении запроса на отправку несуществующего файла всегда отправлять страницу приложения.

Этот ключ обязательно следует указывать, если в публикуемом приложении используется маршрутизатор Browser Router. Если его не указать, при попытке перейти на какой-либо экран приложения путем прямого ввода его интернет-адреса веб-сервер вернет сообщение об ошибке 404.

Если в приложении применяется маршрутизатор Hash Router или навигация вообще не реализована, ключ можно не указывать;

- `--help` — выдача краткой справки о программе:

```
serve --help
```

- `--version` (или `-v`) — выдача номера версии программы.

Следующие ключи используются, если приложение должно работать по протоколу HTTPS:

- `--ssl-cert` <*путь к файлу открытого сертификата SSL*>;
- `--ssl-key` <*путь к файлу закрытого ключа SSL*>;
- `--ssl-pass` <*пароль к закрытому ключу SSL*>.

24.3.2. Публикация на сторонних мощностях

Процесс публикация веб-приложения на мощностях сторонних хостинг-провайдеров (в частности, Google Firebase, Microsoft Azure, GitHub Pages, Heroku и др.) рассмотрен на странице **https://create-react-app.dev/docs/deployment**.

24.4. Выброс React-проекта

React-проект включает в свой состав ряд инструментальных программ, необходимых для его работы и устанавливаемых автоматически при его создании, — в первую очередь это Webpack и Babel. Каждая из этих программ требует указания настроек, которые записываются в особых конфигурационных файлах.

Конфигурационные файлы хранятся в составе библиотеки react-scripts, которая также устанавливается автоматически. Благодаря этому папка React-проекта не загромождается множеством служебных файлов.

Однако такой подход имеет недостаток: невозможность переконфигурирования инструментальных программ под какие-либо специфические условия (например, под работу в старых веб-обозревателях). Поскольку конфигурационные файлы хранятся в составе дополнительных библиотек, исправлять их нежелательно.

Выходом может оказаться процедура *выброса*[1], при которой все без исключения зависимости копируются в манифест React-проекта, а все конфигурационные файлы и часть инструментальных утилит — в папку проекта. В результате появляется возможность внести в конфигурацию проекта необходимые правки.

Выброс выполняется подачей команды:

```
npm run eject
```

В результате в папке проекта появятся следующие папки:

- config — с конфигурационными файлами (включая конфигурационный файл Webpack webpack.config.js);
- scripts — с JavaScript-программами, реализующими команды запуска проекта на выполнение и построения эксплуатационной редакции приложения.

Помимо того, как говорилось ранее, в файл манифеста package.json React-проекта будут занесены все необходимые зависимости.

Проект, подвергнутый выбросу, сохранит свою функциональность. В нем не будет работать лишь команда выброса — по вполне понятной причине.

> ***Внимание!***
>
> **Процедура выброса необратима. После ее выполнения вернуть проект в изначальное состояние невозможно.**

[1] В документации — eject.

Заключение

На этом книга о веб-фреймворке React и программировании веб-приложений с его применением закончена. Мы подробно изучили сам фреймворк, дополнительные библиотеки, существенно облегчающие разработку, различные полезные приемы программирования, инструменты для отладки и процесс публикации готовых приложений. И, наконец, написали вполне функциональное и практически полезное React-приложение для ведения списка планируемых дел, которое можно использовать как основу для создания более развитых и специализированных решений.

React — это невероятное сочетание простоты в освоении, скорости и компактности. Вы, уважаемые читатели, как хотите, но автору нравится эта библиотека!

Судя по всему, фреймворку уготована долгая жизнь. Появившись в 2011 году, став достоянием публики в 2013-м, React лишь наращивал популярность. Многие конкурирующие с ним фреймворки появились, на мгновение блеснули на небосклоне и канули в Лету, а React все так же гордо занимает заслуженное первое место.

Каково будущее этого фреймворка? Безусловно, он будет развиваться, обзаводясь новыми впечатляющими функциональными возможностями, становясь все быстрее, компактнее и удобнее. Уже с уверенностью можно сказать, что компоненты-классы спустя какое-то время станут достоянием истории, полностью уступив место компонентам-функциям. А последние, в свою очередь, наконец-то обзаведутся инструментами для перехвата ошибок.

Сейчас самое время изучить React. Количество проектов, реализованных с его применением, растет, равно как и количество свободных вакансий React-программистов. Так что спешите занять их — иначе вас опередят!

В табл. З.1 приведен перечень дополнительных интернет-ресурсов, посвященных React. Они очень помогут вам в изучении этого замечательного фреймворка.

И напоследок для ознакомления — шесть веб-фреймворков, конкурирующих с React, но не настолько популярных:

- Preact (**https://preactjs.com/**) — альтернатива React, более компактная (программное ядро в сжатом виде занимает всего 3 Кбайт), но и более низкоуровневая. Принципы программирования аналогичны таковым у React. Неплохая альтернатива, если требуется высокая скорость запуска приложения;

Таблица 3.1. Перечень интернет-ресурсов по теме книги и смежным темам

Интернет-адрес	Описание
https://reactjs.dev/[1]	Домашний сайт React. Документация, включая вводный курс для начинающих, новости, блог разработчиков, сведения о поддержке, ряд ссылок на другие тематические ресурсы
https://create-react-app.dev/	Домашний сайт инструментальной утилиты create-react-app, которая служит для создания React-проектов
https://nodejs.org/en/	Домашний сайт Node.js — среды исполнения JavaScript-программ
https://docs.npmjs.com/	Документация по npm — пакетному менеджеру, поставляемому в составе Node.js и предназначенному для установки дополнительных библиотек и инструментальных программ
https://www.npmjs.com/	Репозиторий npm — хранилище дополнительных утилит и инструментальных программ для Node.js. Имеются утилиты и программы, предназначенные для использования с React. Реализован поиск по названию или его части
https://telegram.me/react_js	Два русскоязычных Telegram-канала, посвященных React
http://t.me/react_ru	

- Solid.js (**https://www.solidjs.com/**) — аналог Preact. Предлагает более высокое быстродействие, однако принципы программирования несколько иные;
- Qwik (**https://qwik.dev/**) — аналог Preact. Очень высокое быстродействие, но приемы программирования значительно отличаются от React'овских;
- Vue.js (**https://vuejs.org/**) — более высокоуровневый фреймворк, построенный согласно архитектуре «контроллер — шаблон — оформление». Код каждого компонента делится на три части: собственно программная логика (контроллер), представление, написанное на HTML с вкраплениями специального макроязыка (шаблон), и таблица стилей на обычном CSS (оформление). Хороший выбор для тех, кто любит все раскладывать по полочкам;
- Lit (**https://lit.dev/**) — аналог Vue.js, более компактный и быстрый. Использует подсистему веб-компонентов, встроенную в современные веб-обозреватели;
- Angular (**https://angular.dev/**) — наиболее высокоуровневый, полнофункциональный фреймворк. Включает в себя всё, что необходимо для разработки даже очень сложных приложений: функциональность компонентов, средства навигации, валидации, поддержку анимации, инструменты для взаимодействия с бэкендом и многое другое. Также основан на архитектуре «компонент — шаблон — оформление». Лучший выбор для программирования сложных решений, однако несколько громоздок, сложен в освоении и требует тщательного планирования.

Что ж, на этом — всё! Автор прощается с вами, уважаемые читатели, и напоследок желает успехов в таком увлекательном деле, как веб-программирование.

Владимир Дронов

[1] Русская редакция домашнего сайта React, находящаяся по интернет-адресу **https://ru.reactjs.org/**, больше не обновляется.

ПРИЛОЖЕНИЕ 1

Компоненты-классы

Почти все React-компоненты в настоящее время реализуются в виде функций (вследствие чего их называют *компонентами-функциями*). Компоненты-функции быстры, компактны и не занимают оперативную память.

П1.1. Введение в компоненты-классы

Однако любой компонент в React также может быть реализован в виде класса (*компонент-класс*). У компонентов-классов есть как существенные преимущества перед компонентами-функциями, так и серьезные недостатки.

Преимущества:

- удобные инструменты для перехвата и обработки ошибок, возникающих в содержимом компонента-класса.

  В компонентах-функциях такое нельзя реализовать в принципе;
- возможность хранить произвольные данные в течение всего времени существования компонента — в свойствах представляющего его объекта.

  В компонентах-функциях для этого придется использовать рефы (см. *разд. 4.2.1.2*);
- более развитые инструменты, позволяющие вмешаться в работу компонента для реализации дополнительной функциональности, — волшебные методы;
- возможность управлять компонентами-классами через ссылающиеся на них рефы, вызывая объявленные в компонентах методы.

  Ссылаться на компоненты-функции рефы не могут.

Недостатки:

- невозможность применения хуков в коде компонентов-классов;
- больший объем занимаемой памяти (поскольку экземпляр компонента-класса является объектом класса, он требует дополнительную память для своего хранения);
- более бедный ассортимент доступных программных инструментов.

Многие современные библиотеки предоставляют свои программные инструменты в виде хуков, использовать которые в компонентах-классах нельзя (типичный пример — библиотека React Router, описанная в *главах 11* и *12*).

П1.2. Реализация компонентов-классов

П1.2.1. Объявление класса компонента

Компонент-класс должен быть производным от класса `Component` из модуля `react`. Имя компонента-класса должно начинаться с большой буквы.

В компоненте-классе должны присутствовать, по крайней мере, следующие методы:

- конструктор — требуется для задания изначального состояния компонента и выполнения какой-либо инициализации. В качестве единственного параметра он должен принимать простой объект с пропами, полученными компонентом.

  При объявлении конструктора в самом начале его кода следует вызвать конструктор базового класса, передав ему полученный с параметром объект с пропами, — иначе компонент не заработает.

  Если компонент не имеет состояния и не инициализируется при создании, конструктор объявлять необязательно;
- `render()` — должен возвращать объектную структуру, представляющую содержимое компонента и описанную на языке JSX.

Вот шаблон, согласно которому объявляются компоненты-классы:

```
import { Component } from 'react';

export default class SomeComponent extends Component {
    constructor(props) {
        super(props);
        // Задание изначального состояния
        // Инициализационный код
    }

    render() {
        return (
            // Содержимое компонента, написанное на языке JSX
        );
    }
}
```

В классе компонента также можно объявить произвольные свойства и методы.

П1.2.2. Пропы компонента-класса

Пропы передаются компоненту-классу через единственный параметр его конструктора. По соглашениям этому параметру дают имя `props`.

В коде компонента-класса пропы можно получить, обратившись к свойству `props` компонента. Оно хранит простой объект, свойства которого соответствуют одноименным пропам.

Вот пример обращения к пропу `todos` в коде компонента-класса `TodoList` для вывода хранящегося в нем списка дел:

```
import { Component } from 'react';
export default class TodoList extends Component {
    constructor(props) {
        super(props);
        . . .
    }
    . . .
    render() {
        return (
            . . .
            <table>
                {this.props.todos.map(item => (
                    <tr key={item.key}>
                        . . .
                    </tr>
                ))}
            </table>
            . . .
        );
    }
}
```

Статическое свойство `defaultProps` класса компонента позволяет указать у пропов значения по умолчанию. Значением этого свойства должен быть простой объект, свойства которого должны соответствовать пропам, у которых указываются значения по умолчанию, а значения этих свойств, собственно, станут значениями по умолчанию.

Вот пример указания у пропа `opened` компонента-класса `Spoiler` значения по умолчанию `true`:

```
export default class Spoiler extends Component {
    static defaultProps = { opened: true }
    . . .
}
```

П1.2.3. Состояние компонента-класса

В компоненте-классе можно создать лишь одно состояние, хранящее единственное значение (в отличие от компонента-функции, способного содержать произвольное количество состояний, — см. *разд. 4.1.4*).

Практически всегда в качестве значения состояния у компонента-класса задается простой объект, в свойствах которого хранятся нужные данные (*объект состояния*). Это позволяет занести в состав состояния произвольное количество значений.

Изначальное состояние задается в коде конструктора компонента-класса путем присваивания объекта состояния свойству `state`:

```
constructor(props) {
    super(props);
    this.state = { todos: [], showMenu: false };
    . . .
}
```

Прочитать значение состояния можно из того же свойства `state`. Вот пример получения значения свойства `showMenu` объекта состояния:

```
<div . . . className={this.state.showMenu ?
                      'navbar-menu is-active' :
                      'navbar-menu'}>
    . . .
</div>
```

Для обновления состояния с занесением в него изменившихся данных (и последующим обновлением компонента) следует использовать метод `setState()` компонента-класса, поддерживающий два формата вызова:

```
setState(<функция, изменяющая состояние>[,
         <функция, вызываемая после изменения состояния>=undefined])
setState(<новое состояние>[,
         <функция, вызываемая после изменения состояния>=undefined])
```

Первый формат применяется в большинстве случаев. *Функция, изменяющая состояние*, должна принимать два параметра: текущее состояние и простой объект с пропами (который берется из свойства `props`). Возвращать она должна простой объект, содержащий свойства с новыми значениями, которые следует занести в обновленный объект состояния.

Пример изменения значения свойства `showMenu` объекта состояния:

```
this.setState(state => ({ showMenu: !state.showMenu }));
```

Пример обновления счетчика щелчков на кнопке на величину, заданную в пропе `delta`:

```
this.setState((state, props) => ({
    clickCount: state.clickCount + props.delta
}));
```

Можно обновить состояние (и вызвать обновление компонента), не изменяя записанных в него значений, — для чего достаточно вернуть из *функции, изменяющей состояние*, пустой объект:

```
this.setState(state => ({}));
```

Сразу после обновления состояния и компонента будет вызвана *функция*, указанная во втором параметре метода `setState()`. Она не получает никаких параметров. В ней можно выполнить какие-либо завершающие действия после обновления компонента. Пример:

```
this.setState(state => ({ showMenu: !state.showMenu }),
              () => {
                  const isMenuShown = this.state.showMenu;
                  . . .
              }
);
```

Второй формат вызова метода `setState()` выполняется несколько быстрее первого. Первым параметром следует передать новый объект состояния целиком, со всеми входящими в него свойствами. Пример:

```
this.setState({
    showMenu: !this.state.showMenu,
    todos: this.state.todos
});
```

Этот формат имеет смысл использовать, если в объекте состояния присутствует мало свойств, и в настоящий момент следует обновить бóльшую их часть.

Все операции изменения состояния, осуществляемые вызовами метода `setState()`, не выполняются сразу же, а ставятся в своего рода очередь и производятся после выполнения кода, в котором присутствуют вызовы этого метода (обычно это обработчик события). Такой подход позволяет исключить излишние обновления компонента и тем самым повысить производительность приложения.

> ***Внимание!***
>
> **Метод `setState()` может быть вызван только после монтирования компонента. Если вызвать его до этого момента (например, в конструкторе), возникнет некритическая ошибка.**

П1.2.4. Написание обработчиков событий в компоненте-классе

Обработчики событий в компонентах-классах можно писать в виде:

- обычных методов:

```
import { Component } from 'react';
export default class TodoAdd extends Component {
    . . .
    handleFormSubmit(evt) {
        . . .
    }
    . . .
```

```
    render() {
        return (
            <form onSubmit={this.handleFormSubmit}>
                . . .
            </form>
        );
    }
}
```

Однако в теле метода-обработчика попытка обращения к текущему объекту компонента через переменную `this` приведет к ошибке, поскольку эта переменная будет хранить значение `undefined`. Решить проблему можно следующим образом.

В JavaScript метод класса является свойством, которому присвоен объект класса `Function`, представляющий этот метод. Следовательно, мы можем в конструкторе класса компонента вызвать у представляющего этот метод объекта класса `Function` метод `bind()`, передав ему в качестве параметра ссылку на текущий объект (которая хранится в переменной `this`), присвоить возвращенный методом результат — новый метод, привязанный к текущему объекту, — тому же свойству и использовать новый метод в качестве обработчика события.

Пример:

```
export default class TodoAdd extends Component {
    constructor(props) {
        . . .
        this.handleFormSubmit = this.handleFormSubmit.bind(this);
    }
    . . .
    handleFormSubmit(evt) {
        . . .
    }
    . . .
    render() {
        return (
            <form onSubmit={this.handleFormSubmit}>
                . . .
            </form>
        );
    }
}
```

- стрелочных функций, присвоенных каким-либо свойствам.

  Поскольку в теле стрелочной функции переменная `this` всегда ссылается на текущий объект, привязывать ее к текущему объекту вызовом метода `bind()` не нужно.

Пример:

```
export default class TodoAdd extends Component {
    . . .
    handleFormSubmit = evt => {
        . . .
    }
    . . .
    render() {
        return (
            <form onSubmit={this.handleFormSubmit}>
                . . .
            </form>
        );
    }
}
```

- стрелочных функций, непосредственно привязываемых к тегам (как и в компонентах-функциях — см. *разд. 6.1*).

П1.3. Дополнительные инструменты для написания компонентов-классов

П1.3.1. Рефы в компоненте-классе

Объектный реф в компоненте-классе следует создавать вызовом функции `createRef()` из модуля `react`. Созданный реф следует присвоить какому-либо свойству.

Пример:

```
import { Component, createRef } from 'react';

export default class VideoPlayer extends Component {
    videoRef = createRef()

    render() {
        return (
            . . .
            <video src="/movie.mp4" ref={this.videoRef} />
            <input type="button" value="Пуск"
                   onClick={() => {this.videoRef.current.play()}} />
            <input type="button" value="Пауза"
                   onClick={() => {this.videoRef.current.pause()}} />
            . . .
        );
    }
}
```

Функциональный реф реализуется в виде метода[1]:

```
export default class VideoPlayer extends Component {
    videoRef = null
    setVideoRef = el => {this.videoRef = el}

    render() {
        return (
            . . .
            <video src="/movie.mp4" ref={this.setVideoRef} />
            . . .
        );
    }
}
```

П1.3.2. Получение данных из контекста React

Чтобы подписаться на контекст, в компоненте-классе следует объявить статическое свойство `contextType` и присвоить этому свойству объект контекста, созданный ранее.

Нагрузка контекста будет доступна через свойство `context` компонента.

Вот пример получения массива дел из контекста `TodosContext`:

```
import { Component } from 'react';
import { TodosContext } from './contexts.js';

export default class TodoList extends Component {
    static contextType = TodosContext

    render() {
        return (
            <table>
                {this.context.map(item => (
                    . . .
                ))}
            </table>
        );
    }
}
```

К сожалению, компонент-класс может получать данные лишь из одного контекста.

[1] Который суть частный случай функции.

П1.3.3. Волшебные методы компонента-класса

В *разд. 4.1.1* говорилось, что жизненный цикл компонента делится на три этапа: монтирование, обновление и размонтирование.

В каждый момент жизненного цикла в компоненте-классе вызываются особые методы, называемые *волшебными*[1] (или *методами жизненного цикла*). Они позволяют выполнить на некотором этапе какие-либо действия и в определенных рамках управлять функционированием компонента.

Волшебные методы записываются в объявлении класса компонента. Код, выполняющий необходимые действия и управляющий компонентом, помещается в теле этих методов.

React поддерживает следующие волшебные методы:

- `componentDidMount()` — вызывается сразу после монтирования компонента (но не после последующих обновлений).

  В этом методе уже можно вызывать метод `setState()`, выполняющий обновление состояния.

  В методе `componentDidMount()`, как правило, выполняется подписка на серверные сообщения и устанавливаются соединения по протоколам WebSocket и WebRTC.

  Вот пример установки соединения по протоколу WebSocket:

```
componentDidMount() {
    this.connection = new WebSocket('ws://somesite.ru:2345');
    this.connection.send('HELLO');
    this.connection.send(this.props.id);
}
```

  Метод `componentDidMount()` выполняется до вывода смонтированного компонента на экран. Так что если в этом методе в состояние компонента вносятся какие-либо изменения, компонент будет выведен уже с учетом произведенных изменений;

- `shouldComponentUpdate()` — вызывается перед обновлением компонента в результате изменения его пропов, состояния или нагрузки контекста, на который была выполнена подписка. Позволяет при выполнении заданного условия отменить обновление компонента. Формат:

```
shouldComponentUpdate(<новые пропы>, <новое состояние>[,
                                    <новая нагрузка контекста>=undefined])
```

  Новые пропы передаются в виде простого объекта. Если компонент не подписан ни на один на контекст, последний параметр не передается.

  При выполнении этого метода свойство `props` компонента все еще хранит объект со старыми пропами, свойство `state` — старое состояние, а свойство `context` —

[1] По словам разработчиков React, эти методы получили такое название, поскольку они добавляют компонентам долю «волшебства».

старое значение нагрузки контекста. Благодаря этому, мы можем в теле метода получить доступ как к новым, так и к старым значениям пропов, состояния и нагрузки контекста, чтобы, например, сравнить их.

Метод должен возвращать `true`, если компонент подлежит обновлению, `false` — если компонент обновлять не требуется.

Вот пример метода, указывающего обновить компонент только после получения с пропом `key` другого идентификатора дела:

```
shouldComponentUpdate(nextProps, nextState) {
    return this.props.key !== nextProps.key;
}
```

Если этот метод не объявлен, компонент будет обновляться при любом изменении пропов, состояния или нагрузки контекста (как если бы метод вернул `true`);

- `componentDidUpdate()` — вызывается сразу после обновления компонента в результате изменения его пропов и (или) состояния (но не после монтирования) и позволяет выполнить указанные действия в этот момент:

```
componentDidUpdate(<предыдущие пропы>, <предыдущее состояние>, <снимок>)
```

Предыдущие пропы передаются в виде простого объекта. Третьим параметром передается объект снимка, возвращенный методом `getSnapshotBeforeUpdate()` (будет описан далее), если же этот метод не объявлен, передается значение `null`.

При выполнении этого метода свойство `props` компонента уже хранит объект с новыми пропами, а свойство `state` — новое состояние.

Метод `componentDidUpdate()` вызывается только в том случае, если метод `shouldComponentUpdate()` вернет `true` (т. е. если компонент действительно обновился).

В этом методе обычно выполняются сетевые запросы, основанные на данных, которые получены с обновленными пропами. Также здесь можно внести какие-либо изменения в DOM уже после ее обновления, используя значение, полученное с третьим параметром.

Вот пример загрузки с бэкенда дела с идентификатором, извлекаемым из пропа `key`:

```
async componentDidUpdate(prevProps, prevState, snapshot) {
    if (this.props.key !== prevProps.key) {
        const todo = await loadTodo(this.props.key);
        this.setState(state => ({ todo }));
    }
}
```

Метод `componentDidUpdate()` выполняется до вывода обновленного компонента на экран. Так что если в этом методе в состояние компонента вносятся какие-либо изменения, компонент будет выведен уже с учетом произведенных изменений;

- `componentWillUnmount()` — вызывается непосредственно перед размонтированием компонента и позволяет выполнить какие-либо завершающие действия.

  В этом методе нельзя вызывать метод `setState()`, поскольку компонент уже не будет обновляться.

  В методе `componentWillUnmount()` обычно отписываются от серверных сообщений и разрывают ранее установленные соединения по протоколам WebSocket и WebRTC.

  Пример:

```
componentWillUnmount() {
    this.connection.close();
}
```

- `getSnapshotBeforeUpdate()` — вызывается непосредственно перед внесением изменений в DOM, выполняемым в процессе обновления компонента. Позволяет получить какую-либо информацию из DOM перед ее изменением. Формат вызова:

  `getSnapshotBeforeUpdate(`*<предыдущие пропы>*`, `*<предыдущее состояние>*`)`

  Предыдущие пропы передаются в виде простого объекта.

  При выполнении этого метода свойство `props` компонента уже хранит объект с новыми пропами, а свойство `state` — новое состояние.

  Метод способен возвращать снимок текущего состояния DOM, которым может быть произвольное значение (например, позиция прокрутки у какого-либо элемента страницы). Этот снимок далее будет передан методу `componentDidUpdate()` с третьим параметром. Если возврат снимка не предусматривается, все же следует вернуть из метода значение `null`.

  Вот пример использования этого метода в компоненте, выводящем прокручиваемый список сообщений чата:

```
import { Component, createRef } from 'react';

export default class MessageList extends Component {
    constructor(props) {
        super(props);
        . . .
        // Создаем реф, ссылающийся на блок со списком сообщений
        // (рефы описаны в разд. П1.3.2)
        this.listRef = createRef();
    }
    . . .
    getSnapshotBeforeUpdate(prevProps, prevState) {
        // Если в список добавились новые сообщения...
        if (prevProps.list.length < this.props.list.length) {
            // ...возвращаем текущую позицию прокрутки списка по вертикали
            const list = this.listRef.current;
            return list.scrollHeight - list.scrollTop;
```

```
        } else
            // В противном случае возвращаем null
            return null;
    }

    componentDidUpdate(prevProps, prevState, snapshot) {
        // Если была получена предыдущая позиция прокрутки списка
        // сообщений, возвращенная методом getSnapshotBeforeUpdate()...
        if (snapshot !== null) {
            // ...прокручиваем список на эту позицию
            const list = this.listRef.current;
            list.scrollTop = list.scrollHeight - snapshot;
        }
    }

    render() {
        return (
            <div ref={this.listRef}>
                // Здесь выводятся сообщения из списка
            </div>
        );
    }
}
```

- `getDerivedStateFromProps()` — статический метод, вызывается перед монтированием и перед обновлением компонента. Позволяет после создания или изменения состояния дополнительно добавить в него значения из полученных пропов. Формат записи:

  `getDerivedStateFromProps(<`*новые пропы*`>, <`*предыдущее состояние*`>)`

  Новые пропы передаются в виде простого объекта.

  Метод должен вернуть простой объект со значениями, подлежащими добавлению в объект состояния. Возвращаемый им объект аналогичен тому, который возвращается функцией, задаваемой в вызове метода `setState()` (см. *разд. П1.2.3*). Если никаких значений в состояние добавлять не требуется, метод должен вернуть `null`.

  Будучи статическим, этот метод не имеет доступа к свойствам и методам объекта текущего компонента.

  Метод предусмотрен для крайне редких случаев, когда значения пропов зачем-либо хранятся в состоянии компонента. Вот пример:

```
static getDerivedStateFromProps(props, state) {
    if (props.key === state.key)
        return null;
    else
        return { key: props.key };
}
```

При монтировании компонента-класса его методы вызываются в следующем порядке:

- конструктор;
- `getDerivedStateFromProps()`;
- `render()`;
- `componentDidMount()`.

При обновлении компонента-класса порядок вызова методов таков:

- `getDerivedStateFromProps()`;
- `shouldComponentUpdate()`.

Следующие методы вызываются, только если метод `shouldComponentUpdate()` вернет `true`:

- `render()`;
- `getSnapshotBeforeUpdate()`;
- `componentDidUpdate()`.

При размонтировании компонента вызывается лишь метод `componentWillUnmount()`.

П1.3.4. Принудительное обновление компонента-класса

В редких случаях может потребоваться обновить компонент-класс принудительно. Это можно выполнить, вызвав у компонента метод `forceUpdate()`:

```
forceUpdate([<функция, вызываемая после обновления компонента>=undefined])
```

Задаваемая *функция* не должна ни принимать параметров, ни возвращать результата. Если она не указана, после обновления компонента ничего не произойдет.

Следует помнить, что метод `shouldComponentUpdate()` при вызове метода `forceUpdate()` не выполняется.

П1.4. Средства для обработки ошибок в компонентах-классах. Перехватчики

Вероятно, единственное существенное преимущество компонентов-классов перед компонентами-функциями — возможность реализовать обработку ошибок, возникающих в их содержимом (в частности, во вложенных в них компонентах). В компонентах-функциях обрабатывать такие ошибки в принципе невозможно[1].

Обработка ошибок в компоненте-классе реализуется в телах следующих волшебных методов:

[1] Разработчики React явно поторопились, объявляя компоненты-классы устаревшими. Впрочем, возможно, они собираются вскоре реализовать подобную функциональность в компонентах-функциях.

- `getDerivedStateFromError(<исключение>)` — статический, вызывается после возникновения ошибки в коде содержимого текущего компонента. В качестве параметра получает объект *исключения*, описывающий возникшую ошибку. Должен возвращать простой объект, содержащий свойства с новыми значениями, которые будут занесены в обновляемый объект состояния.

  Метод вызывается в процессе рендеринга, поэтому в его теле нельзя обращаться к внешним API (например, выводить сообщение в консоль веб-обозревателя);

- `componentDidCatch()` — также вызывается после возникновения ошибки в коде содержимого текущего компонента. Формат вызова:

  ```
  componentDidCatch(<исключение>, <стек вызова>)
  ```

  Стек вызова представляется простым объектом со свойством `componentStack`, хранящим содержимое стека вызова. Это содержимое представлено в виде строки, разбитой на подстроки с помощью символов `/n`, поэтому его можно без проблем вывести в консоль.

  В теле этого метода обычно выполняется обращение к каким-либо внешним API (например, консоли веб-обозревателя) с целью вывести сообщение о возникшей ошибке. Если этого делать не требуется, метод можно не объявлять.

Перехватчик — это компонент-класс, специально предназначенный для обработки ошибок, которые возникают в коде компонентов-потомков. В случае возникновения ошибки перехватчик обычно отображает на экране соответствующее уведомление и выводит сообщение об ошибке в консоли веб-обозревателя.

Использование перехватчиков позволяет, во-первых, централизовать обработку ошибок, возникающих в разных компонентах, а во-вторых, предотвратить аварийную остановку приложения при возникновении ошибки.

Листинг П1.1 показывает код перехватчика `ErrorCatcher`, который в случае возникновения ошибки в каком-либо из потомков выводит на страницу соответствующее оповещение, а в консоли — сведения об ошибке и стек вызова.

Листинг П1.1. Перехватчик `ErrorCatcher`

```
import { Component } from 'react';

export default class ErrorCatcher extends Component {
    constructor(props) {
        super(props);
        this.state = { isError: false };
    }

    static getDerivedStateFromError(exc) {
        return { isError: true };
    }

    componentDidCatch(exc, errorInfo) {
        console.log(exc);
```

```
        console.log(errorInfo.componentStack);
    }

    render() {
        if (this.state.isError)
            return <p>Возникла ошибка.</p>;
        else
            return this.props.children;
    }
}
```

Пример использования этого перехватчика в стартовом модуле:

```
import ErrorCatcher from './ErrorCatcher.js';
. . .
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(
    <React.StrictMode>
        <ErrorCatcher>
            <App />
        </ErrorCatcher>
    </React.StrictMode>
);
```

За кулисами

В состав провайдера маршрутизации, предоставляемого библиотекой React Router (см. *разд. 11.2.5*), входит компонент `RenderErrorBoundary`, предназначенный для обработки возникающих в целевых компонентах ошибок (подробности — в *разд. 21.2*). Этот компонент является перехватчиком.

П1.5. Повышение производительности компонентов-классов. Лишние обновления и их устранение

Предположим, что компонент приложения перечня дел `TodoList` использует для вывода каждой позиции вложенный компонент `TodoItem`, имеющий такой код:

```
import { Component } from 'react';
export default class TodoItem extends Component {
    render() {
        return (
            <tr>
                <td>
                    {item.done && <del>{this.props.item.title}</del>}
                    {!item.done && this.props.item.title}
                </td>
```

```
                <td>
                    <button . . . >&#9745;</button>
                </td>
                <td>
                    <button . . . >&#9746;</button>
                </td>
            </tr>
        );
    }
}
```

Для вывода компонентов `TodoItem` в компоненте `TodoList` применяется такой код:

```
<table>
    {props.list.map((item) => (
        <TodoItem key={item.key} item={item} . . . />
    ))}
</table>
```

Теперь предположим, что пользователь пометил какое-либо дело как выполненное, нажав на соответствующую кнопку. В этом случае состояние компонента `TodoList` изменится, и этот компонент будет обновлен. И соответственно будут обновлены его компоненты-потомки `TodoItem`.

Проблема заключается в том, что при обновлении компонента-родителя обновляются *все* его потомки, а не только тот, в котором выводится изменившееся дело. Эта проблема носит название *лишнего обновления* и при наличии большого количества потомков может привести к серьезной потере производительности.

Существуют два способа устранить лишние обновления у компонентов-классов: сложный и простой. Рассмотрим их именно в таком порядке (почему — будет ясно позже).

П1.5.1. Явное подавление лишних обновлений

Сложный способ заключается в том, чтобы перед собственно обновлением компонента проверять, изменились ли данные, полученные компонентом от родителя посредством пропов, и обновлять компонент, только если они изменились. Сложным он называется потому, что для его реализации придется писать дополнительный программный код.

Проверку на изменение полученных данных следует производить в волшебном методе `shouldComponentUpdate()` (см. *разд. П1.3.3*). Вот пример кода, производящего такую проверку:

```
export default class TodoItem extends Component {
    . . .
    shouldComponentUpdate(nextProps, nextState) {
        return nextProps.item !== this.props.item;
    }
    . . .
}
```

Чтобы этот код работал, помимо изменения самого массива дел, придется также заменить элемент массива, хранящий сведения об этом деле, на другой, содержащий исправленные данные. Вот код метода `setDone()`, выполняющий эти действия:

```
export default class App extends Component {
    . . .
    setDone(key) {
        // Создаем копию массива дел
        const newData = [...this.state.data];
        // Ищем в нем дело с указанным идентификатором
        const deedIndex = newData.findIndex(
            current => current.key === key
        );
        // Если дело найдено, заменяем элемент массива, хранящий сведения
        // о нем, на другой, исправленный, в котором дело помечено
        // как выполненное
        if (deedIndex > -1)
            newData[deedIndex] = { ...newData[deedIndex], done: true };
        // Не забываем обновить состояние
        this.setState(state => ({ data: newData }));
    }
    . . .
}
```

П1.5.2. Создание чистых компонентов

Как правило, для подавления лишних обновлений в методе `shouldComponentUpdate()` компонента-класса пишут довольно-таки тривиальный код, сравнивающий старые и новые значения пропов. Чтобы упростить жизнь программистам, разработчики React предоставили так называемый *чистый компонент*, который:

- обновляется — только если получит от родителя с пропами изменившиеся данные;
- уже содержит в себе метод `shouldComponentUpdate()`, сравнивающий старое и новое значения каждого из пропов, а также старое и новое значения каждого из свойств объекта состояния, и разрешает обновление только в том случае, если старое и новое значения какого-либо из пропов или свойств объекта состояния различаются.

Функциональность чистого компонента реализована в классе `PureComponent` из модуля `react`. Чтобы превратить какой-либо компонент-класс в чистый, достаточно сделать его производным от этого класса.

Вот пример чистого компонента `TodoItem` (метод `shouldComponentUpdate()` в этом классе больше не нужен, и его можно удалить):

```
import { PureComponent } from 'react';
export default class TodoItem extends PureComponent {
    . . .
```

```
    shouldComponentUpdate(nextProps, nextState) { . . . }
    . . .
}
```

П1.6. Отладка компонентов-классов

Расширение React Developer Tools предоставляет дополнительные инструменты для отладки компонентов-классов.

В правой части вкладки **Components** окна расширения React Developer Tools (см. *разд. 23.3.1*), в числе прочих сведений о компоненте, выбранном в списке левой части этой вкладки, отображаются также следующие разделы:

- **state** — состояние компонента-класса. Можно разворачивать сложные значения, изменять их и копировать все состояние в формате JSON в буфер обмена;
- **context** — значение, получаемое компонентом из контекста. Их можно только разворачивать и копировать в буфер обмена.

На вкладке **Components** окна настроек этого расширения (см. *разд. 23.3.3*) в раскрывающемся списке из группы элементов управления **Hide components where**, задающем характеристику скрываемого из списка компонента, также доступны для выбора следующие разновидности компонентов, которые требуется скрыть:

- **class** — компонент-класс;
- **function** — компонент-функция.

ПРИЛОЖЕНИЕ 2

Устаревший API библиотеки React Router

Библиотека React Router, реализующая навигацию в React-приложениях (см. *главы 11* и *12*), все еще поддерживает устаревшие программные инструменты для описания маршрутов, оставшиеся «в наследство» от версий 6.0–6.3[1]. Код, использующий устаревшие инструменты, до сих пор можно встретить во всевозможных старых разработках.

> ***Внимание!***
>
> Вывод гиперссылок, перенаправление, получение значений URL-параметров и сведений о совпавшем маршруте при использовании списка маршрутов, созданного посредством устаревшего API, производится с применением знакомых нам инструментов, описанных в *разд. 11.3–11.4*.
>
> Инструменты, описанные в *разд. 11.5* и *главе 12*, в старом API недоступны. Так что создавать загрузчики и воздействия, применять средства для обработки веб-форм, фетчеры и блокировать уход при использовании старого API невозможно.

Все компоненты, составляющие устаревший API, объявлены в модуле `react-router-dom`.

П2.1. Базовые инструменты маршрутизации

- Отдельный маршрут — представляется знакомым нам компонентом `Route` (см. *разд. 11.2.1*).

  Список маршрутов помещается в коммутатор.

- *Коммутатор* — компонент, выполняющий поиск совпавшего маршрута и выводящий указанный в нем целевой компонент.

  Коммутатор реализуется компонентом `Routes`. Как говорилось ранее, в нем размещается список маршрутов, которые он должен обрабатывать.

  Компонент `Routes` поддерживает проп `caseSensitive`. Если ему дано значение `true`, при сравнении путей будет учитываться регистр символов, которыми они

---

[1] Новый API, описанный в *главах 11–12*, появился в версии 6.4 этой библиотеки.

набраны, если `false` — *не* будет учитываться. Значение пропа по умолчанию: `false`.

Коммутатор размещается в одном из доступных маршрутизаторов.

- Маршрутизатор:
  - Browser Router (см. *разд. 11.2.4.1*) — представляется компонентом `BrowserRouter`;
  - Hash Router (см. *разд. 11.2.4.2*) — представляется компонентом `HashRouter`.

  Оба компонента поддерживают проп `basename`, который задает префикс, добавляемый к путям из гиперссылок, создаваемых React Router. Префикс указывается в виде строки — с начальным слешем, но без конечного.

  Внутри маршрутизатора помещается коммутатор со списком маршрутов.

Для вывода сообщения об ошибке 404 в конце списка маршрутов записывается маршрут с шаблонным путем * (звездочка) и специально написанным компонентом, который и будет выводить сообщение об ошибке.

Сам маршрутизатор помещается в том месте интерфейса приложения, в котором должны выводиться целевые компоненты (обычно это компонент приложения `App`). Все компоненты гиперссылок должны находиться внутри маршрутизатора — иначе они не будут работать. Как правило, внутри маршрутизатора помещают весь интерфейс приложения.

Провайдер маршрутизации в этом случае не используется.

Вот пример списка маршрутов, последовательно вложенный в коммутатор и маршрутизатор и помещенный в компонент приложения:

```
import { BrowserRouter, Routes, Route, NavLink } from 'react-router-dom';
export default function App(){
    . . .
    return (
        <BrowserRouter>
            <nav>
                . . .
                <NavLink to="/">Перечень дел</NavLink>
                <NavLink to="/add">Добавить дело</NavLink>
                . . .
            </nav>
            <main>
                <Routes>
                    <Route path="/" element={<TodoList />} />
                    <Route path="/:key" element={<TodoDetail />} />
                    <Route path="/add" element={<TodoAdd />} />
                    . . .
                </Routes>
            </main>
        </BrowserRouter>
    );
}
```

В стартовом модуле в этом случае выводят компонент приложения:

```
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(
    <React.StrictMode>
        <App />
    </React.StrictMode>
);
```

П2.2. Создание иерархического списка маршрутов средствами устаревшего API

При использовании устаревшего API также можно создавать иерархические списки маршрутов (см. *разд. 11.2.2.2.2*), например:

```
import { BrowserRouter, Routes, Route } from 'react-router-dom';
. . .
<BrowserRouter>
    . . .
    <Routes>
        <Route path="/">
            <Route path="" element={<TodoList />} />
            <Route path=":key" element={<TodoDetail />} />
            <Route path="add" element={<TodoAdd />} />
            <Route path="user" element={<UserLayout />}>
                <Route index element={<Profile />} />
                <Route path="login" element={<Login />} />
                <Route path="logout" element={<Logout />} />
                <Route path="register" element={<Register />} />
            </Route>
        </Route>
    </Routes>
</BrowserRouter>
```

В разметочных компонентах, указанных в узловых маршрутах, допускается использовать выпуски (см. *разд. 11.2.3*).

ПРИЛОЖЕНИЕ 3

Описание файлового архива

Файловый архив, сопровождающий книгу, размещен на сайте издательства «БХВ» по интернет-адресу: **https://zip.bhv.ru/9785977520362.zip**. Ссылка на него доступна и со страницы книги на сайте **https://bhv.ru/**.

Список папок и файлов, имеющихся в архиве:

- apache-http-server-config — папка с конфигурационным файлом для веб-сервера Apache HTTP Server, необходимым для публикации React-приложения (подробности — в *разд. 24.3.1.1*);
- todos-1-0 — папка с программным кодом версии 1.0 учебного веб-приложения для ведения списка планируемых дел, разрабатываемого во вводном курсе (*глава 1*);
- todos-2-0 — папка с программным кодом версии 2.0 учебного веб-приложения для ведения списка планируемых дел, разрабатываемого во вводном курсе (*глава 2*);
- readme.txt — текстовый файл с описанием файлового архива и инструкциями по запуску учебного веб-приложения.

Предметный указатель

_

__REDUX_DEVTOOLS_EXTENSION__() 403
__REDUX_DEVTOOLS_EXTENSION_COMPOSE__() 404

A

abort() 312
action 180, 182, 239, 240
action() 386
actions 303
addCase() 296
addDefaultCase() 298
addMatcher() 297
align 369
allowOffset 338
applyMiddleware() 277
areMergedPropsEqual 289
areOwnPropsEqual 289
areStatePropsEqual 289
areStatesEqual 288
arg 309
array() 340
as 315, 375, 377
AttentionSeeker 358
AuthProvider 348
Await 249, 383

B

Babel 31
basename 219, 440
bindActionCreators() 270
boolean() 339
Bounce 355
Browser Router 218
BrowserRouter 440

C

camelCase 156
cascade 357
caseSensitive 213, 225, 232, 439
cast() 345
childClassName 356
children 128, 233
Children 129
childStyle 356
className 223, 356, 362, 367
columnCount 366
columnIndex 365, 366, 369, 370
columnWidth 366, 369
combineReducers() 271
compact() 341
component 320, 328, 333
Component 212, 422
Component() 386
componentDidCatch() 434
componentDidMount() 429
componentDidUpdate() 430
componentStack 434
componentWillUnmount() 431
condition 308
configureStore() 299
connect() 283
Consumer 169
context 283, 290, 428
contextType 428
count() 129
createAction() 294
createAsyncThunk() 305
createBrowserRouter() 218
createContext() 166
createDispatchHook() 293
createHashRouter() 219
createPortal() 139
createReducer() 295
createRef() 427
createRoutesFromElements() 214
createSelector() 273, 312
createSelectorHook() 293
createSlice() 301
createStore() 265
createStoreHook() 293
CSS-модуль 155
current 136
currentLocation 230

D

damping 357
dangerouslySetInnerHTML 151
data 182, 244, 245, 251, 350, 361, 366
date() 340
datetime() 338
default() 336
defaultProps 423
defer() 249
delay 356
devTools 300
devToolsExtension() 301
direction 354, 355
dirty 324
disabled 373
dispatch 286, 306
dispatch() 267
dispatchConditionRejection 308
duration 355

E

effect 358
element 212
element() 386
email() 337
enableReinitialize 314
encType 242
end 224, 232
enhancers 300

ПРИЛОЖЕНИЕ 3

Описание файлового архива

Файловый архив, сопровождающий книгу, размещен на сайте издательства «БХВ» по интернет-адресу: **https://zip.bhv.ru/9785977520362.zip**. Ссылка на него доступна и со страницы книги на сайте **https://bhv.ru/**.

Список папок и файлов, имеющихся в архиве:

- apache-http-server-config — папка с конфигурационным файлом для веб-сервера Apache HTTP Server, необходимым для публикации React-приложения (подробности — в *разд. 24.3.1.1*);
- todos-1-0 — папка с программным кодом версии 1.0 учебного веб-приложения для ведения списка планируемых дел, разрабатываемого во вводном курсе (*глава 1*);
- todos-2-0 — папка с программным кодом версии 2.0 учебного веб-приложения для ведения списка планируемых дел, разрабатываемого во вводном курсе (*глава 2*);
- readme.txt — текстовый файл с описанием файлового архива и инструкциями по запуску учебного веб-приложения.

Предметный указатель

_

__REDUX_DEVTOOLS_EXTENSION__() 403
__REDUX_DEVTOOLS_EXTENSION_COMPOSE__() 404

A

abort() 312
action 180, 182, 239, 240
action() 386
actions 303
addCase() 296
addDefaultCase() 298
addMatcher() 297
align 369
allowOffset 338
applyMiddleware() 277
areMergedPropsEqual 289
areOwnPropsEqual 289
areStatePropsEqual 289
areStatesEqual 288
arg 309
array() 340
as 315, 375, 377
AttentionSeeker 358
AuthProvider 348
Await 249, 383

B

Babel 31
basename 219, 440
bindActionCreators() 270
boolean() 339
Bounce 355
Browser Router 218
BrowserRouter 440

C

camelCase 156
cascade 357
caseSensitive 213, 225, 232, 439
cast() 345
childClassName 356
children 128, 233
Children 129
childStyle 356
className 223, 356, 362, 367
columnCount 366
columnIndex 365, 366, 369, 370
columnWidth 366, 369
combineReducers() 271
compact() 341
component 320, 328, 333
Component 212, 422
Component() 386
componentDidCatch() 434
componentDidMount() 429
componentDidUpdate() 430
componentStack 434
componentWillUnmount() 431
condition 308
configureStore() 299
connect() 283
Consumer 169
context 283, 290, 428
contextType 428
count() 129
createAction() 294
createAsyncThunk() 305
createBrowserRouter() 218
createContext() 166
createDispatchHook() 293
createHashRouter() 219
createPortal() 139
createReducer() 295
createRef() 427
createRoutesFromElements() 214
createSelector() 273, 312
createSelectorHook() 293
createSlice() 301
createStore() 265
createStoreHook() 293
CSS-модуль 155
current 136
currentLocation 230

D

damping 357
dangerouslySetInnerHTML 151
data 182, 244, 245, 251, 350, 361, 366
date() 340
datetime() 338
default() 336
defaultProps 423
defer() 249
delay 356
devTools 300
devToolsExtension() 301
direction 354, 355
dirty 324
disabled 373
dispatch 286, 306
dispatch() 267
dispatchConditionRejection 308
duration 355

E

effect 358
element 212
element() 386
email() 337
enableReinitialize 314
encType 242
end 224, 232
enhancers 300

ПРИЛОЖЕНИЕ 3

Описание файлового архива

Файловый архив, сопровождающий книгу, размещен на сайте издательства «БХВ» по интернет-адресу: **https://zip.bhv.ru/9785977520362.zip**. Ссылка на него доступна и со страницы книги на сайте **https://bhv.ru/**.

Список папок и файлов, имеющихся в архиве:

- apache-http-server-config — папка с конфигурационным файлом для веб-сервера Apache HTTP Server, необходимым для публикации React-приложения (подробности — в *разд. 24.3.1.1*);
- todos-1-0 — папка с программным кодом версии 1.0 учебного веб-приложения для ведения списка планируемых дел, разрабатываемого во вводном курсе (*глава 1*);
- todos-2-0 — папка с программным кодом версии 2.0 учебного веб-приложения для ведения списка планируемых дел, разрабатываемого во вводном курсе (*глава 2*);
- readme.txt — текстовый файл с описанием файлового архива и инструкциями по запуску учебного веб-приложения.

Предметный указатель

_

__REDUX_DEVTOOLS_EXTENSION__() 403
__REDUX_DEVTOOLS_EXTENSION_COMPOSE__() 404

A

abort() 312
action 180, 182, 239, 240
action() 386
actions 303
addCase() 296
addDefaultCase() 298
addMatcher() 297
align 369
allowOffset 338
applyMiddleware() 277
areMergedPropsEqual 289
areOwnPropsEqual 289
areStatePropsEqual 289
areStatesEqual 288
arg 309
array() 340
as 315, 375, 377
AttentionSeeker 358
AuthProvider 348
Await 249, 383

B

Babel 31
basename 219, 440
bindActionCreators() 270
boolean() 339
Bounce 355
Browser Router 218
BrowserRouter 440

C

camelCase 156
cascade 357
caseSensitive 213, 225, 232, 439
cast() 345
childClassName 356
children 128, 233
Children 129
childStyle 356
className 223, 356, 362, 367
columnCount 366
columnIndex 365, 366, 369, 370
columnWidth 366, 369
combineReducers() 271
compact() 341
component 320, 328, 333
Component 212, 422
Component() 386
componentDidCatch() 434
componentDidMount() 429
componentDidUpdate() 430
componentStack 434
componentWillUnmount() 431
condition 308
configureStore() 299
connect() 283
Consumer 169
context 283, 290, 428
contextType 428
count() 129
createAction() 294
createAsyncThunk() 305
createBrowserRouter() 218
createContext() 166
createDispatchHook() 293
createHashRouter() 219
createPortal() 139
createReducer() 295
createRef() 427
createRoutesFromElements() 214
createSelector() 273, 312
createSelectorHook() 293
createSlice() 301
createStore() 265
createStoreHook() 293
CSS-модуль 155
current 136
currentLocation 230

D

damping 357
dangerouslySetInnerHTML 151
data 182, 244, 245, 251, 350, 361, 366
date() 340
datetime() 338
default() 336
defaultProps 423
defer() 249
delay 356
devTools 300
devToolsExtension() 301
direction 354, 355
dirty 324
disabled 373
dispatch 286, 306
dispatch() 267
dispatchConditionRejection 308
duration 355

E

effect 358
element 212
element() 386
email() 337
enableReinitialize 314
encType 242
end 224, 232
enhancers 300

ПРИЛОЖЕНИЕ 3

Описание файлового архива

Файловый архив, сопровождающий книгу, размещен на сайте издательства «БХВ» по интернет-адресу: **https://zip.bhv.ru/9785977520362.zip**. Ссылка на него доступна и со страницы книги на сайте **https://bhv.ru/**.

Список папок и файлов, имеющихся в архиве:

- apache-http-server-config — папка с конфигурационным файлом для веб-сервера Apache HTTP Server, необходимым для публикации React-приложения (подробности — в *разд. 24.3.1.1*);
- todos-1-0 — папка с программным кодом версии 1.0 учебного веб-приложения для ведения списка планируемых дел, разрабатываемого во вводном курсе (*глава 1*);
- todos-2-0 — папка с программным кодом версии 2.0 учебного веб-приложения для ведения списка планируемых дел, разрабатываемого во вводном курсе (*глава 2*);
- readme.txt — текстовый файл с описанием файлового архива и инструкциями по запуску учебного веб-приложения.

Предметный указатель

_

__REDUX_DEVTOOLS_EXTENSION__() 403
__REDUX_DEVTOOLS_EXTENSION_COMPOSE__() 404

A

abort() 312
action 180, 182, 239, 240
action() 386
actions 303
addCase() 296
addDefaultCase() 298
addMatcher() 297
align 369
allowOffset 338
applyMiddleware() 277
areMergedPropsEqual 289
areOwnPropsEqual 289
areStatePropsEqual 289
areStatesEqual 288
arg 309
array() 340
as 315, 375, 377
AttentionSeeker 358
AuthProvider 348
Await 249, 383

B

Babel 31
basename 219, 440
bindActionCreators() 270
boolean() 339
Bounce 355
Browser Router 218
BrowserRouter 440

C

camelCase 156
cascade 357
caseSensitive 213, 225, 232, 439
cast() 345
childClassName 356
children 128, 233
Children 129
childStyle 356
className 223, 356, 362, 367
columnCount 366
columnIndex 365, 366, 369, 370
columnWidth 366, 369
combineReducers() 271
compact() 341
component 320, 328, 333
Component 212, 422
Component() 386
componentDidCatch() 434
componentDidMount() 429
componentDidUpdate() 430
componentStack 434
componentWillUnmount() 431
condition 308
configureStore() 299
connect() 283
Consumer 169
context 283, 290, 428
contextType 428
count() 129
createAction() 294
createAsyncThunk() 305
createBrowserRouter() 218
createContext() 166
createDispatchHook() 293
createHashRouter() 219
createPortal() 139
createReducer() 295
createRef() 427
createRoutesFromElements() 214
createSelector() 273, 312
createSelectorHook() 293
createSlice() 301
createStore() 265
createStoreHook() 293
CSS-модуль 155
current 136
currentLocation 230

D

damping 357
dangerouslySetInnerHTML 151
data 182, 244, 245, 251, 350, 361, 366
date() 340
datetime() 338
default() 336
defaultProps 423
defer() 249
delay 356
devTools 300
devToolsExtension() 301
direction 354, 355
dirty 324
disabled 373
dispatch 286, 306
dispatch() 267
dispatchConditionRejection 308
duration 355

E

effect 358
element 212
element() 386
email() 337
enableReinitialize 314
encType 242
end 224, 232
enhancers 300

ensure() 338, 340
error 309, 329, 350
ErrorBoundary 381
ErrorBoundary() 386
errorElement 380, 383
errorElement() 386
ErrorMessage 320
errors 324, 330, 344, 350
estimatedColumnWidth 370
estimatedItemSize 365
estimatedRowHeight 369
excludeEmptyString 337
extraReducers 302

F

Fade 353
fallback 192, 385
fallbackElement 220
fetcherKey 246
field 328
Field 315, 328
FieldArray 331
FirebaseAppProvider 347
firebaseConfig 347
FirestoreProvider 348
FixedSizeGrid 365
FixedSizeList 360
Flip 354
flushSync() 141
for 147
forceUpdate() 433
forEach() 130
form 329, 333
Form 239, 245, 315
formAction 180, 245, 252
formData 245, 252
formEncType 252
Formik 313, 346
formMethod 245, 252
forwardRef() 175
fraction 356
Fragment 125
fulfilled 309

G

generatePath() 223
getDefaultEnhancers() 300
getDefaultMiddleware() 299
getDerivedStateFromError() 434
getDerivedStateFromProps() 432
getFieldProps() 328
getKey 255
getSnapshotBeforeUpdate() 431
getState 306, 308
getState() 267
Google Firebase 67
Google Firebase SDK 76

H

handle 251, 253
handleBlur() 327
handleChange() 326
handleReset() 327
handleSubmit() 327
hash 221, 228
Hash Router 219
HashRouter 440
height 361, 366
Hinge 355
historyAction 231
horizontalScrollDirection 367
href 374
htmlFor 147
HTTP-ошибка 382

I

id 238, 251, 410
idField 351
import() 384
index 215, 361
initialError 329
initialErrors 322
initialScrollLeft 367
initialScrollOffset 362
initialScrollTop 367
initialState 302
initialStatus 314
initialTouched 322, 329
initialValue 329
initialValues 313
innerElementType 362, 367
innerRef 316, 363, 368
insert() 331
integer() 339
is 342
isActive 224
isDefaultPrevented() 160
isPending 224
isPropagationStopped() 160
isScrolling 361, 366
isSubmitting 324, 330
isValid 324
isValid() 345
isValidating 324, 330
isValidSync() 345
itemCount 361
itemData 362, 366
itemKey 362, 366
itemSize 361, 365

J

JackInTheBox 355
json 252
json() 236, 341, 343
JSX 29, 146

K

key 130, 152, 228, 243, 247

L

layout 361
lazy 387
lazy() 385
length() 337, 340
lessThan() 339
Link 221
load() 244
loader 237
loader() 386
location 231, 252
lowercase() 338

M

map() 130
match() 295
matches() 337
matchPath() 232
max() 337, 339, 340
memo() 143
message 309, 337, 338
meta 329
method 182, 240
middleware 299
min() 337, 339, 340
moreThan() 339
move() 332

N

name 301, 315, 320, 328, 331, 333
nativeEvent 159
navigate 246
NavLink 223
negative() 339
nextLocation 230
notOneOf() 336
npx 23
nullable() 336
number() 339

O

object() 341
ObservableStatus 349
of() 340
onBlur 329
onChange 329
oneOf() 336
onError 374
onItemRendered 362, 367
onLoad 373
onlyChild() 130
onRender 410
onReset 330
onScroll 363, 367
onSubmit 318
onVisibilityChange 356
originalValue 335, 343
otherwise 342
outerElementType 362, 367
outerRef 363, 368
Outlet 218
overscanColumnCount 367
overscanColumnStartIndex 367
overscanColumnStopIndex 367
overscanCount 362
overscanRowCount 367
overscanRowStartIndex 367
overscanRowStopIndex 367
overscanStartIndex 362
overscanStopIndex 362

P

params 232, 236, 251
PascalCase 120
path 212, 232
pathname 221, 228, 232, 251
pattern 233
payload 258
pending 182, 309
pop() 332
positive() 339
precedence 373, 374
precision 338
preconnect() 378
prefetchDNS() 378
preinit() 377
preinitModule() 378
preload() 375
preloadedState 300
preloadModule() 376
prepare 302
preventScrollReset 256
proceed() 231
Profiler 410
props 130, 423
Provider 283
PureComponent 437
push() 331

R

React Awesome Reveal 353
React Developer Tools 390
React Redux 282
React Router 210
React Window 360
ReactFire 347
redirect() 237
reducer 299, 302, 303
reducers 301
Redux 262
Redux DevTools 403
Redux DevTools Extension 404
Redux Thunk 280
Redux Toolkit 294
ref 136, 138, 174
rejected 309
rejectWithValue 307
relative 222, 226, 240
remove() 332
render 320
render() 108, 422
replace 221, 226, 240
replace() 332
replaceReducer() 273
request 235
requestId 306, 309
requestStatus 309
required() 336, 337
Reselect 273
reset() 231
resetAfterColumnIndex() 370
resetAfterIndex() 365
resetAfterIndices() 370
resetAfterRowIndex() 370
resetForm() 326, 330
resolve 249
Roll 355
Rotate 354
round() 339
Route 212, 439
router 220
RouterProvider 220
Routes 439
rowCount 366
rowHeight 366, 369
rowIndex 365, 366, 369, 370

S

scrollDirection 363
scrollLeft 368, 369
scrollOffset 363
ScrollRestoration 255
scrollTo() 364, 369
scrollToItem() 364, 369
scrollTop 368, 369
scrollUpdateWasRequested 363, 368
sdk 348
search 221, 228
selectors 302, 303
serve 415
setErrors() 325
setFieldError() 325
setFieldTouched() 325
setFieldValue() 324
setState() 424
setStatus() 326
setSubmitting() 326
settled 309
setTouched() 326
setValues() 324
shape() 341
shouldComponentUpdate() 429
shouldForceUpdate 370
signal 307
signedIn 350
SigninCheckResult 350
Slide 355
startTransition() 195
state 221, 226, 228, 231, 240, 244, 245, 252, 424
status 324, 330, 350
store 283
StrictMode 388
string() 337
style 156, 224, 356, 361, 362, 366, 367
submit() 246
submitCount 324, 330
submitForm() 326
subscribe() 268
Suspense 192, 385
swap() 332
SyntheticBaseEvent 159

T

test() 343
text 252
then 342
thunk() 280
to 221
toArray() 130
touched 324, 329, 330
transform() 344
triggerOnce 356
trim() 338
truncate() 339
type 130, 258, 316

U

unshift() 331
unwrap() 311
uppercase() 338
url() 337
URL-параметр 58, 211
use() 167, 193
useActionData() 243
useAsyncError() 383
useAsyncValue() 250
useBeforeUnload() 229
useBlocker() 230
useCallback() 172
useContext() 167
useDatabase() 351
useDatabaseListData() 351
useDatabaseObjectData() 352
useDebugValue() 402
useDeferredValue() 198
useDispatch() 292
useEffect() 187
useFetcher() 243
useFetchers() 248
useFirebaseApp() 348
useFormStatus() 182
useId() 141
useImperativeHandle() 175
useIsScrolling 362, 366
useLayoutEffect() 190
useLoaderData() 237
useLocation() 228
useMatch() 233
useMatches() 251
useMemo() 144
useNavigate() 225
useNavigation() 252
useOptimistic() 200
useParams() 226
user 350
useReducer() 259
useRef() 136, 137
useRouteError() 380
useRouteLoaderData() 238
useSearchParams() 227
useSelector() 290
useSigninCheck() 349
useState() 131
useStore() 292
useSubmit() 241
useSyncExternalStore() 191
useTransition() 197
useUser() 350
uuid() 337

V

validate 319, 321
validate() 344
validateField() 325
validateForm() 325
validateOnBlur 322
validateOnChange 321, 331
validateOnMount 322
validateSync() 345
ValidationError 344
validationSchema 346
value 167, 328, 335
values 323, 330
VariableSizeGrid 369
VariableSizeList 364
verticalScrollDirection 368
visibleColumnStartIndex 367
visibleColumnStopIndex 367
visibleRowStartIndex 367
visibleRowStopIndex 367
visibleStartIndex 362
visibleStopIndex 362

W

Web Vitals 409
Webpack 31
when() 342
width 361, 366

Y

Yup 334

Z

Zoom 355

А

Атрибут 87

Б

Бандл 31
Блокировка ухода 230
Бэкенд 13

В

Валидация 96, 185
Веб-приложение 13
◊ бессерверное 67
◊ одностраничное 14, 25
◊ серверное 70
Веб-страница приложения 103, 106
Веб-фреймворк клиентский 14
Величина с отложенным изменением 198
Виртуальная DOM 124
Влияние 257, 263
Воздействие React 180
◊ полнофункциональное 183
◊ упрощенное 180
Воздействие React Router 54, 238
Временна́я диаграмма 396
Выброс 417
Выпуск 49, 218

Г

Гиперссылка активная 50, 221
Гость 94
График продолжительности рендеринга 398

Д

Документ 87
◊ корневой 87

Ж

Жизненный цикл 121

З

Зависимость 144
Загрузка по запросу 384
Загрузчик 53, 235

К

Ключ 152
Кнопка-гамбургер 51

Колбэк-реф 137
Коллекция 87
Коммит 396
Коммутатор 439
Композиция 202
Компонент 25, 120, 131
◊ вложенный 26
◊ высшего порядка 202
◊ класс 421
◊ приложения 26, 120
◊ разметочный 217
◊ связанный 282
◊ функция 421
◊ целевой 46, 211
◊ чистый 437
◊ экран 25
Контейнер 30
Контекст 166

Л

Линтер 110

М

Маршрут 46, 211
◊ дочерний 47, 215
◊ конечный 216
◊ корневой 215
◊ параметризованный 58, 211
◊ разметочный 229
◊ родительский 47, 215
◊ совпавший 46, 211
◊ узловой 216
Маршрутизатор 46, 211
Маршрутизация 211
Массив элементов управления 331
Мемоизация 142
Меню навигации 30, 51
Местоположение 228
Метод
◊ волшебный 429
◊ жизненного цикла 429
Модуль стартовый 26, 108
Монитор кода 112
Монтирование 39, 121

Н

Навигация 46, 210
Нагрузка 166

О

Обновление 121
◊ лишнее 436
◊ низкоприоритетное 194
◊ оптимистичное 200
Обозначение бренда 30
Объект
◊ состояния 424
◊ хранения 315
Отправитель
◊ влияний 270
◊ клиентских запросов 58
Отслеживание кода 112
Ошибка
◊ 404 64
◊ критическая 113
◊ некритическая 113

П

Папка проекта 22, 102
Переменная окружения 107, 114
Перенаправление 56, 212
◊ рефа 174
Перехватчик 434
Переходник 280
Подъем данных 36, 164, 170
Портал 139, 162
Посетитель текущий 77
Посредник 275
Построитель веб-форм 313
Потомок 26
Потребитель контекста 169
Преобразователь
◊ React 257, 258
◊ Redux 262, 264
◊ составной 271
◊ среза 271
Приостановщик 192
Провайдер
◊ базы данных 348
◊ контекста 166
◊ маршрутизации 48, 211, 220
◊ подсистемы разграничения доступа 348
◊ серверного веб-приложения 347
◊ хранилища 283
Проект 22, 102
◊ серверный 68
Промежуточный обработчик 175
Проп 31, 122, 149, 164
Профилировщик 395
Путь 46, 88, 210
◊ текущий 46, 211
◊ целевой 50
◊ шаблонный 46, 211

Р

Разделение кода 384
Размонтирование 39, 122
Расширитель хранилища 277
Реактивные данные 38, 123
Рендеринг 124
◊ первоначальный 124
◊ повторный 124
Рендер-проп 173
Реф 136
◊ объектный 136
◊ функциональный 137
Родитель 26

С

Сборщик 31
Сброс стилей 109
Связка 31, 111
Сегмент 211
Селектор 263, 266
Сеттер
◊ GET-параметров 227
◊ состояния 39, 131, 132
Слот 128
Слушатель 263, 268
Снимок 90
Создатель влияний 269
Сосед 26
Состояние 38, 131
◊ хранилища 262
Список маршрутов 211
◊ иерархический 215
◊ простой 214
Спуск данных 31, 123, 164
◊ простой 165
◊ сквозной 165
Срез 271
Статус веб-формы 314
Строгий режим 388
СУБД документная 87
Схема 334

Т

Таблица стилей
◊ компонента 154
◊ приложения 109
Тег
◊ компонента 34
◊ корневой 30, 124
Транслятор 31

У

Условный вывод 151

Ф

Файл
◊ окружения 115
▫ локальный 115
Фетчер 80, 243
Фрагмент 125
Фреймворк 14
Фронтенд 13
Функция
◊ диспетчер 260
◊ очистки 81, 189

Х

Хлебные крошки 253
Хранилище 262, 265
Хук 39, 122

Ч

Чанк 111

Ш

Шаблон 103

Э

Экран 14, 25
◊ главный 211
Элемент управления
◊ посещенный 320
Эффект 83, 187

Я

Якорь 35, 141

React 19
Разработка веб-приложений на JavaScript

Простота и скорость клиентской веб-разработки

Книга посвящена разработке веб-приложений на языке JavaScript с применением популярного клиентского веб-фреймворка React 19. В ней учтены все нововведения, появившиеся в React 19 и новых версиях сопутствующих библиотек.

Дается вводный курс для новичков в React, содержащий пошаговое описание разработки несложного веб-приложения — списка планируемых дел. Описываются базовые инструменты и приемы React-программирования: создание и настройка проекта React-приложения, написание компонентов, обработка событий, передача данных между компонентами, программирование веб-форм и компоненты высшего порядка. Рассматриваются дополнительные библиотеки, которые пригодятся любому React-разработчику: React Router (навигация), Redux (централизованное хранилище данных), React Redux и Redux Toolkit (упрощение разработки и использования централизованного хранилища), Formik (быстрая разработка веб-форм), Yup (валидация введенных данных), React Awesome Reveal (создание анимационных эффектов) и др. Далее ведется рассказ о работе с метаданными, разделении кода, обработке ошибок, отладочных инструментах (включая расширения веб-обозревателя React Developer Tools и Redux DevTools), публикации готового веб-приложения с использованием веб-серверов Apache HTTP Server и serve.



Дронов Владимир Александрович, профессиональный программист, писатель и журналист, работает с компьютерами с 1987 года. Автор около 50 популярных компьютерных книг, в том числе «Node.js, Express, MongoDB и React. 23 урока для начинающих», «JavaScript. 32 урока для начинающих», «Django 4. Практика создания веб-сайтов на Python», «Laravel 9. Быстрая разработка веб-сайтов на PHP», «Python 3 и PyQt 6. Разработка приложений» и др.



Электронное приложение-архив, содержащее полный код описанного в книге веб-приложения, можно скачать по ссылке https://zip.bhv.ru/9785977520362.zip, а также со страницы книги на сайте bhv.ru

ISBN 978-5-9775-2036-2





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

bhv®